Jürgen Kehrel

Apple-Assembler lernen

Band 1: Einführung in die Assembler-Programmierung

Hüthig

6502/65C02

Jürgen Kehrel · Apple-Assembler lernen

Jürgen Kehrel

Apple-Assembler lernen

Band 1: Einführung in die Assembler-Programmierung des 6502/65C02

Dr. Alfred Hüthig Verlag Heidelberg

Diejenigen Bezeichnungen von im Buch genannten Erzeugnissen, die zugleich eingetragene Warenzeichen sind, wurden nicht besonders kenntlich gemacht. Es kann also aus dem Fehlen der Markierung ® nicht geschlossen werden, daß die Bezeichnung ein freier Warenname ist. Ebensowenig ist zu entnehmen, ob Patente oder Gebrauchsmusterschutz vorliegt.

Als Ergänzung zu diesem Buch ist gesondert lieferbar:
»Begleitdiskette zu Apple-Assembler lernen, Bd. 1«
ISBN 3-7785-1243-9

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Kehrel, Jürgen:
Apple-Assembler lernen/Jürgen Kehrel. –
Heidelberg: Hüthig

Bd. 1. Einführung in die Assembler-Programmierung
des 6502/65C02. – 1986
ISBN 3-7785-1151-3

© 1986 Dr. Alfred Hüthig Verlag GmbH Heidelberg
Printed in Germany
Satz, Druck und Bindung: Laub GmbH, Elztal-Dallau

0. Vorwort

Sie haben gehört, daß Assembler wahnsinnig schnell ist. Sie wissen, daß Assembler sehr kompakte Programme erzeugt. Sie haben aber auch gehört, daß Assembler schwierig zu erlernen sei und haben deshalb bisher erst gar nicht so richtig damit angefangen. Sie haben in vielen Zeitschriften Assemblerlistings gesehen und gehofft, sie mögen auf Diskette erscheinen. Oder Sie haben Byte für Byte die HEX-Dumps abgetippt ohne zu wissen, was Sie da eigentlich eingeben, und hinterher den großen Frust erlebt, wenn Sie sich verschrieben haben und nichts läuft.
Sie mußten feststellen, daß viele interessante Programme in Assembler geschrieben sind und Dinge tun, die Sie mit Basic nicht schaffen. Sie stehen Assembler mit einem Gefühl zwischen Bewunderung und Abscheu gegenüber. **SIE** sind gerade richtig, um am Kurs „Apple-Assembler lernen" teilzunehmen. Dieser Kurs hilft Ihnen systematisch, die Assemblerprogrammierung des Apple II mit allen Tricks und Finessen zu erlernen. Dabei wird sowohl im Buch als auch bei der Software der 65C02-Prozessor des Apple IIc und des neuesten (enhanced) Apple IIe voll unterstützt.
Ein **Komfort-Assembler** für 6502- und 65C02-Code sowie ein anschaulicher **6502-Simulator** sind auf der Begleitdiskette enthalten, so daß Sie keine weitere Software kaufen müssen, um alle Teile dieses Kurses auch praktisch durchzuführen.

Sie sollten folgende **Vorkenntnisse** besitzen:
- Normale Bedienung des Apple II
- mittlere Fähigkeiten in der Basic-Programmierung oder einer anderen Hochsprache. Sie brauchen aber *kein* Basic-Experte zu sein.

An **Hardware** werden benötigt:
- Apple II mit Applesoft im ROM (IIplus, IIe oder IIc)
- 1 Diskettenlaufwerk
- Monitor oder (notfalls) Fernseher.

Außerdem sind zwei Leerdisketten und die „DOS 3.3 System Master"-Diskette von Apple erforderlich.

Heidelberg, Oktober 1985 Dr. Jürgen B. Kehrel

0. Vorwort

Inhalt

0. Vorwort 5

1. Einleitung 11

2. Der langsame Start 13

3. Jetzt machen wir Fortschritte 15
Lektion 1: Wie der Blitz 15
Lektion 2: Das rasende Her(t)z des Apple 21
Lektion 3: Wieviele Bits bitte? 23
Lektion 4: HEX ist keine Hexerei 27
Lektion 5: Vorzeichen zum Vorzeigen 30
Lektion 6: Mit festem Wohnsitz 32
Lektion 7: Ein Spaziergang im Monitor 35
Lektion 8: Vom Kopf zum Chip 39
Lektion 9: Legale Tauschgeschäfte 41
Lektion 10: IDUS – So tun als ob 45
Lektion 11: Nehmen, Geben und Tauschen 48
Lektion 12: Auf und Ab mit Rundenzähler 54
Lektion 13: Ein Karussell mit Ausgang 57
Lektion 14: 1 + 1 = 3, auch das kann sein 60
Lektion 15: Borgen bringt Sorgen 63
Lektion 16: Zeige Flagge beim Vergleichen 67
Lektion 17: Schleife vor und zurück 70
Lektion 18: Hast du da noch Töne 73
Lektion 19: Wir setzen Zeichen 78
Lektion 20: Ein guter Abgang 81
Lektion 21: Links herum und rechts herum 84
Lektion 22: Kreisverkehr 87
Lektion 23: Keine wundersame Vermehrung 89
Lektion 24: Teile und herrsche 92

Lektion 25: Gesetzt oder nicht? 95
Lektion 26: Ein wenig Logik 97
Lektion 27: BIT für Bits 100
Lektion 28: Erst voll, aber schon übergelaufen 102
Lektion 29: Tief im Keller sitz' ich drin 104
Lektion 30: Wo komme ich her? 107
Lektion 31: Eine kleine Unterbrechung 112
Lektion 32: Restbestände 116
Lektion 33: Flaggenparade 118
Lektion 34: A weiß, daß B weiß, wo C wohnt 121
Lektion 35: Ein Zuhause für mein Programm 129

4. Der 6502/65C02 Befehlssatz 133

5. ASSESSOR 182
Einführung 182
Systemvoraussetzungen 183
Der Aufbau von ASSESSOR 184
Der Start von ASSESSOR 184
Das Format des Quellcodes 185
Ausdrücke und Adressierungsarten 187
Der Kommando-Modus 190
Der Editier-Modus 198
Die Pseudo-Opcodes von ASSESSOR 199
Fehlermeldungen 206
Besondere Hinweise 207
Interna 208
Speicherbelegung unter ASSESSOR 209
Hilfsprogramme für ASSESSOR 211
KOMMEX 211
ASSESSOR-Filer 212
6. IDUS – Der Debugger und Simulator 215
Einführung 215
Systemvoraussetzungen 216
Der Start von IDUS 217
Die Simulatorgraphik 217
Das Dialogfenster 218
Der Simulator 222
Spezielles 223
Speicherbelegung 224

Anhang 1: HEX – DEZ Umwandlung 225

Anhang 2: ASCII-Tabelle 226

Anhang 3: Bildschirmdarstellung 227

Anhang 4: 6502/65C02-Befehlstabelle 229

Register 230

1. Einleitung

Programmieren ist eine reizvolle geistige Herausforderung und eine nützliche Tätigkeit zugleich. Sie benötigen Ihre logischen Fähigkeiten bei der Analyse eines Problems und Ihre kreativen Fähigkeiten zu dessen Lösung. Programmieren erfordert von Ihnen sowohl Präzision als auch Vorstellungsvermögen. Sie müssen den Rahmen einer Computersprache exakt einhalten und sich strikt an die vorgegebenen Regeln halten, da der Rechner uneinsichtig jede, auch die kleinste Abweichung zurückweist. Sie können ein Problem aber meistens auf viele verschiedene Arten lösen, und es liegt an Ihnen, ob Sie den kürzesten, den verständlichsten oder den schnellsten Weg beschreiten. Mit etwas Ehrgeiz werden Sie versuchen, kurze, schnelle *und* verständliche Programme zu schreiben.

Genauso, wie Sie nicht von heute auf morgen sprechen oder singen gelernt haben, werden Sie nicht über Nacht zu einem Programmierprofi. Aber mit diesem Buch und den vielen Programmen auf den Begleitdisketten werden Sie schnell Fortschritte machen.

So wie Sie eine Fremdsprache am besten lernen, wenn Sie sie sprechen und hören, so lernen Sie die Assemblerprogrammierung am leichtesten, wenn Sie praktische Übungen an Ihrem Rechner vollziehen. Die Diskette zum Buch entspricht der Schallplatte oder Kassette zum Fremdsprachenkurs.

Um Ihnen überschaubare Aufgaben zu bieten, ist jeder Band in einzelne Lektionen eingeteilt, die aufeinander aufbauen und Sie befähigen, stetig schwierigere Probleme anzugehen. Diese Einteilung bedeutet nicht, daß Sie immer gleich eine ganze Lektion bearbeiten müssen. Sie selbst bestimmen das Tempo und damit, ob Sie eine Woche, einen Tag oder eine Stunde mit einem Abschnitt verbringen. Erst wenn Sie alles verstanden haben, sollten Sie fortfahren. Wenn Sie bereits gute Vorkenntnisse besitzen, können Sie auch bei einer späteren Lektion „einsteigen". Tauchen dann Fragen auf, führt Sie das Stichwortverzeichnis an die Stelle zurück, an der das Problem behandelt wurde.

Nun wünsche ich Ihnen viel Spaß, denn „computern“ ist auch ein Abenteuer auf unbekannten Pfaden. Nehmen sie den Spaß aber nicht zu ernst: auch Computer haben den berühmten Ausschaltknopf.

2. Der langsame Start

Bevor wir mit dem eigentlichen Kurs beginnen, sind ein paar Vorbereitungen notwendig. Die originale Begleitdiskette zu diesem Band ist schreibgeschützt (die Kerbe links ist zugeklebt). Das bedeutet, daß Sie den Apple nicht benutzen können, um Informationen auf dieser Diskette zu verändern. Sie können die Diskette jedoch mechanisch beschädigen (zerkratzen, verknicken usw.) oder mit einem Magneten (Motor, Lautsprecher usw.) unbrauchbar machen. Es ist deshalb eine gute Praxis, nur mit einer Kopie zu arbeiten und das Original an einem sicheren Ort aufzubewahren.

Diskette und Buch sind urheberrechtlich geschützt. Für Ihren persönlichen Bedarf ist es erlaubt, eine Kopie der Diskette anzufertigen. Sie dürfen jedoch keine Kopien erstellen, um diese an beliebige andere Personen zu verschenken, zu verleihen oder zu verkaufen.

Die Begleitdiskette ist im DOS 3.3 Format beschrieben. Aus urheberrechtlichen Gründen enthält sie jedoch nicht das Betriebssystem der Firma Apple. Sie ist deshalb nicht „bootfähig".
Kopieren Sie die Begleitdiskette mit dem Programm „COPYA" Ihrer DOS 3.3 Systemdiskette (System Master) auf eine neue Diskette. Legen sie dazu den „System Master" in Laufwerk 1. Schalten Sie den Apple ein, falls das noch nicht geschehen ist, und tippen Sie „RUN COPYA" gefolgt von <RETURN>. (Anm.: Ausdrücke in spitzen Klammern bedeuten in diesem Buch, daß Sie die entsprechende(n) Taste(n) drücken. Geben Sie **nicht** die einzelnen Buchstaben ein! <RETURN> ist 1 Tastendruck, ebenso <ESCAPE>. Auch die einen Befehl einschließenden Anführungszeichen geben Sie nicht ein.)
Beantworten Sie jetzt die Fragen entsprechend Ihrem Geräteaufbau. Übertragen Sie, nachdem Sie eine Kopie angefertigt haben, auf diese das Betriebssystem mit dem Programm „MASTER CREATE". Legen Sie dazu wieder die „System Master"-Diskette in Laufwerk 1 und tippen Sie „BRUN MASTER

CREATE <RETURN>“ ein. Folgen Sie den Anweisungen. Der Name des Startprogramms (GREETING PROGRAM) ist „APPLE-ASSEMBLER LERNEN BD.1“.

Jetzt ist Ihre Begleitdiskette für alle weiteren Verwendungen vorbereitet.

3. Jetzt machen wir Fortschritte

Lektion 1: Wie der Blitz

Nach so aufwendigen Vorarbeiten sollen Sie jetzt mit ein paar Demonstrationen verwöhnt werden. Wir wollen an drei Beispielen den Geschwindigkeitsunterschied zwischen BASIC und Maschinensprache einmal akustisch und optisch erfahren. Starten Sie Ihre Sicherheitskopie (Sie haben doch eine gemacht?) der Begleitdiskette: Apple einschalten, sonst „PR#6 <RETURN>". (Dies setzt voraus, daß Sie Ihr Diskettenlaufwerk wie üblich in Steckleiste (Slot) 6 betreiben.) Es erscheint zunächst ein Titelbild, das nach Tastendruck wieder verschwindet und dem „CATALOG" der Diskette Platz macht. Starten Sie das Beispiel Nummer 1A mit „RUN BSP1A <RETURN>". Es enthält das folgende kleine Applesoft-Programm:

```
10  LAUT =  - 1059
20  CALL LAUT
30  FOR I = 0 TO 32767: NEXT
40  CALL LAUT
50  END
60  REM   BEISPIEL NR. 1A
```

Ein „Piep" des Lautsprechers kündigt den Beginn des Programmablaufs an. Danach zählt eine Schleife von 0 bis 32767 hoch, bevor wieder ein Ton erzeugt wird. Bitte haben Sie etwas Geduld mit BASIC, es dauert einige Sekunden. Was es mit der Zahl 32767 auf sich hat, werden wir erst etwas später kennenlernen. Mit „RUN <RETURN>" können Sie das Programm erneut starten.
Das Beispiel Nummer 1B führt genau dieselben Aufgaben aus. Starten Sie es mit „BRUN BSP1B <RETURN>". Jetzt liegen die Töne deutlich näher beieinander. Das zeigt an, um wieviel schneller das Maschinenprogramm zählt,

verglichen mit dem Applesoft-Beispiel. Mit „CALL 768 <RETURN>“ können Sie es erneut starten.

```
                        1   ********************************
                        2   *      BEISPIEL NR. 1B         *
                        3   ********************************
                        4   *
                        5           ORG  $300
                        6   *
                        7   LAUT    EQU  $FBDD      ; = -1059
                        8   *
0300: 20 DD FB          9   START   JSR  LAUT       ; Ton ausgeben
0303: A9 00            10           LDA  #$00       ; Zähler
0305: 85 06            11           STA  $06        ; initialisieren
0307: 85 07            12           STA  $07
0309: E6 06            13   FOR     INC  $06        ; Schleife hochzählen
030B: D0 FC            14           BNE  FOR
030D: E6 07            15           INC  $07
030F: 10 F8            16   NEXT    BPL  FOR        ; Ende erreicht?
0311: 20 DD FB         17           JSR  LAUT       ; Ja, Ton ausgeben
0314: 60               18           RTS             ; ENDE
```

Sie brauchen den Assemblercode jetzt noch nicht zu verstehen. aber nach ein paar weiteren Lektionen können Sie noch einmal zurückblättern und sich den Ablauf ansehen.

Zwei weitere Beispiele: In Nummer 2A und 2B wird jeweils von 0 bis 255 gezählt und dann ein „Tick“ am Lautsprecher erzeugt. (Wegen der Hardware-Eigenschaften hören Sie nur jeden zweiten „Tick“ beim 6502, aber jeden „Tick“ beim 65C02). Vergleichen Sie wieder die BASIC- und Assemblerversion. Das Beispiel 2B läuft so schnell, daß Sie die einzelnen „Ticks“ nicht mehr hören können, sondern einen durchgehenden Ton vernehmen.

```
10  FOR I = 0 TO 255
20  NEXT I
30  A =  PEEK (49200)
40  GOTO 10
50  REM  BEISPIEL NR. 2A
```

```
                    1    ********************************
                    2    *       BEISPIEL NR. 2B        *
                    3    ********************************
                    4    *
                    5            ORG  $300
                    6    *
0300: A0 00         7    START   LDY  #$00
0302: C8            8    FOR     INY             ; Zählschleife
0303: D0 FD         9    NEXT    BNE  FOR
0305: AD 30 C0      10           LDA  $C030      ; = 49200
0308: 4C 02 03      11   GOTO    JMP  FOR
```

Übrigens: BSP2A können Sie beenden, indem Sie <CTRL-C> (Control-Taste *und* C gleichzeitig) betätigen, für BSP2B müssen Sie <RESET> oder <CTRL-RESET> drücken. „RUN <RETURN>" bzw. „CALL 768 <RETURN>" startet die Programme erneut, wenn sie noch im Speicher sind.

Nach den Ohren wollen wir jetzt auch den Augen etwas bieten. Die Beispiele 3A und 3B zeigen eine grafische Umsetzung. Ein Rechteck wird ständig mit akustischer Untermalung umlaufen. Stop und Neustart geschehen wie im Beispiel 2.

```
 10  CLEAR : HGR2
 20  HCOLOR= 6
 30  HPLOT 92,61 TO 184,61 TO 184,129 TO 92,129 TO 92,61
 40  HCOLOR= 3: GOSUB 100: HCOLOR= 0: GOSUB 100
 50  HCOLOR= 3: GOSUB 110: HCOLOR= 0: GOSUB 110
 60  HCOLOR= 3: GOSUB 120: HCOLOR= 0: GOSUB 120
 70  HCOLOR= 3: GOSUB 130: HCOLOR= 0: GOSUB 130
 80  GOTO 40
 100 Y = 59: FOR X = 88 TO 189: HPLOT X,Y: NEXT :A =  PEEK
     (49200): RETURN
 110 X = 189: FOR Y = 59 TO 131: HPLOT X,Y: NEXT :A =  PEEK
     (49200): RETURN
 120 Y = 131: FOR X = 189 TO 88 STEP  - 1: HPLOT X,Y: NEXT :A =
     PEEK (49200): RETURN
 130 X = 88: FOR Y = 131 TO 59 STEP  - 1: HPLOT X,Y: NEXT :A =
     PEEK (49200): RETURN
140  REM    BEISPIEL NR. 3A
```

```
                        1   ********************************
                        2   *       BEISPIEL NR. 3B        *
                        3   ********************************
                        4   *
                        5           ORG   $300
                        6   *
                        7   HCOLORZ  EQU   $E4       ; HCOLOR
                        8   HGR2     EQU   $F3D8     ; HGR2
                        9   HGLIN    EQU   $F53A     ; HPLOT TO
                        10  HPLOT    EQU   $F457     ; HPLOT
                        11  *
0300: 20 D8 F3          12  START    JSR   HGR2      ; HGR2 einschalten
0303: A2 D5             13          LDX   #$D5        ; HCOLOR=6
0305: 86 E4             14          STX   HCOLORZ
0307: A9 3D             15          LDA   #61       ; Y=61 Koordinaten
0309: A2 5C             16          LDX   #92       ; X=92 Rechteck
030B: A0 00             17          LDY   #0
030D: 20 57 F4          18          JSR   HPLOT     ; Punkt zeichnen
0310: A0 3D             19          LDY   #61
0312: A9 B8             20          LDA   #184
0314: A2 00             21          LDX   #0
0316: 20 3A F5          22          JSR   HGLIN     ; Linie zeichnen
0319: A0 81             23          LDY   #129
031B: A9 B8             24          LDA   #184
031D: A2 00             25          LDX   #0
031F: 20 3A F5          26          JSR   HGLIN     ; Linie zeichnen
0322: A0 81             27          LDY   #129
0324: A9 5C             28          LDA   #92
0326: A2 00             29          LDX   #0
0328: 20 3A F5          30          JSR   HGLIN     ; Linie zeichnen
032B: A0 3D             31          LDY   #61
032D: A9 5C             32          LDA   #92
032F: A2 00             33          LDX   #0
0331: 20 3A F5          34          JSR   HGLIN     ; Linie zeichen
                        35  *
                        36  *
0334: A2 7F             37  RUND     LDX   #$7F      ; HCOLOR = 3
0336: 86 E4             38          STX   HCOLORZ
0338: 20 6F 03          39          JSR   SEITE1    ; Seite umranden
033B: A2 00             40          LDX   #0        ; HCOLOR = 0
033D: 86 E4             41          STX   HCOLORZ
033F: 20 6F 03          42          JSR   SEITE1    ; wieder löschen
0342: A2 7F             43          LDX   #$7F      ; HCOLOR = 3
0344: 86 E4             44          STX   HCOLORZ
0346: 20 86 03          45          JSR   SEITE2    ; Seite umranden
0349: A2 00             46          LDX   #0        ; HCOLOR = 0
034B: 86 E4             47          STX   HCOLORZ
034D: 20 86 03          48          JSR   SEITE2    ; wieder löschen
0350: A2 7F             49          LDX   #$7F
0352: 86 E4             50          STX   HCOLORZ
0354: 20 9C 03          51          JSR   SEITE3
0357: A2 00             52          LDX   #0
0359: 86 E4             53          STX   HCOLORZ
035B: 20 9C 03          54          JSR   SEITE3
```

```
035E: A2 7F      55             LDX  #$7F
0360: 86 E4      56             STX  HCOLORZ
0362: 20 B3 03   57             JSR  SEITE4
0365: A2 00      58             LDX  #0
0367: 86 E4      59             STX  HCOLORZ
0369: 20 B3 03   60             JSR  SEITE4
036C: 4C 34 03   61             JMP  RUND        ;  ganz umrundet
                 62   *
036F: A2 58      63   SEITE1    LDX  #88         ; Punkt an Punkt
0371: 86 06      64             STX  $06         ; setzen
0373: A9 3B      65   LOOP      LDA  #59
0375: A0 00      66             LDY  #0
0377: 20 57 F4   67             JSR  HPLOT
037A: E6 06      68             INC  $06
037C: A6 06      69             LDX  $06
037E: E0 BE      70             CPX  #190
0380: 90 F1      71             BLT  LOOP
0382: 2C 30 C0   72             BIT  $C030       ; Ton ausgeben
0385: 60         73             RTS
                 74   *
0386: A9 3B      75   SEITE2    LDA  #59         ; s.o.
0388: A2 BD      76   LOOP1     LDX  #189
038A: A0 00      77             LDY  #0
038C: 48         78             PHA
038D: 20 57 F4   79             JSR  HPLOT
0390: 68         80             PLA
0391: 18         81             CLC
0392: 69 01      82             ADC  #1
0394: C9 84      83             CMP  #132
0396: 90 F0      84             BLT  LOOP1
0398: 2C 30 C0   85             BIT  $C030
039B: 60         86             RTS
                 87   *
039C: A2 BD      88   SEITE3    LDX  #189        ; s.o.
039E: 86 06      89             STX  $06
03A0: A9 83      90   LOOP2     LDA  #131
03A2: A0 00      91             LDY  #0
03A4: 20 57 F4   92             JSR  HPLOT
03A7: C6 06      93             DEC  $06
03A9: A6 06      94             LDX  $06
03AB: E0 58      95             CPX  #88
03AD: B0 F1      96             BGE  LOOP2
03AF: 2C 30 C0   97             BIT  $C030
03B2: 60         98             RTS
                 99   *
03B3: A9 83      100  SEITE4    LDA  #131        ; s.o.
03B5: A2 58      101  LOOP3     LDX  #88
03B7: A0 00      102            LDY  #0
03B9: 48         103            PHA
03BA: 20 57 F4   104            JSR  HPLOT
03BD: 68         105            PLA
03BE: 38         106            SEC
03BF: E9 01      107            SBC  #1
03C1: C9 3B      108            CMP  #59
```

```
03C3: B0 F0      109\          BGE  LOOP3
03C5: 2C 30 C0   110\          BIT  $C030
03C8: 60         111           RTS
```

Die Assemblerprogramme sind nicht nur viel schneller als ihre BASIC-Kollegen, sie benötigen auch weniger Speicherplatz, wie folgende Tabelle zeigt (BASIC ohne REM-Zeile).

	Applesoft	Assembler	Differenz
BSP 1	57 Bytes	21 Bytes	63% kürzer
BSP 2	42 Bytes	11 Bytes	74% kürzer
BSP 3	305 Bytes	201 Bytes	34% kürzer

Auch Compiler können Applesoft nur mäßig „Beine machen". Der ganz in BASIC geschriebene und dann mit TASC compilierte Assembler „ASSYST" aus dem „Visible Computer"-Paket benötigt incl. der Zeit zum Laden des Quellcodes von Diskette ca. 9 Minuten und 50 Sekunden für die 199 Zeilen des „Mini Assemblers". Der 100% in Assembler geschriebene „ASSESSOR" braucht, wiederum incl. Ladezeit, für denselben Quellcode 7 Sekunden. (Beide Assembler ohne paralleles Listing auf dem Bildschirm.)

Zum Abschluß noch ein kleiner Vorschlag, der Ihnen vielleicht auch bei Ihrer BASIC-Programmierung hilft: lassen Sie im Beispiel 2A einmal das „I" nach „NEXT" weg, so daß die Zeile nur noch „20 NEXT" lautet. Auch Applesoft kann man etwas auf die Sprünge helfen.

Lektion 2: Das rasende Her(t)z des Apple

In Ihrem Apple ist eine Menge Intelligenz vorhanden. Sie besteht im wesentlichen aus den elektronischen Bausteinen (Hardware) und den fest eingebauten Programmen (Firmware). Das Zentrum des Apple ist sein Microprozessor, eine integrierte Schaltung mit der Bezeichnung 6502 oder 65C02. Daneben besitzt er Speicherplätze sowie Ein- und Ausgabeelemente. Einige davon sind im Gerät eingebaut (z.B. die Tastatur), andere müssen extra angeschlossen werden (z.B. Monitor).

Zwischen den verschiedenen Teilen des Apple werden fortwährend Informationen in Form von elektrischen Strömen verschoben. Bei einem digitalen Rechner kennen wir nur zwei Zustände: entweder ist der Strom auf einer Leitung an oder aus. Sie können das mit einer Lampe vergleichen, die Sie ein- und ausschalten. Wenn Strom in einer Leitung fließt, sagen wir, daß sie den logischen Zustand „1" hat. Ist der Strom abgeschaltet, nennen wir diesen logischen Zustand „0". Also: **An = 1 , Aus = 0.**

Der Informationsfluß läuft nicht immer kontinuierlich, sondern nur in bestimmten Zeitintervallen, den Takten. Diese Takte werden von einem Generator im Apple hergestellt und an alle Bauteile weitergegeben. Der Generator ist so etwas wie der Steuermann bei einem großen Ruderboot, der die Schlagzahl angibt, damit alle im gleichen Rhythmus ihre Ruder bewegen. Genau so müssen die Bauteile des Apple synchronisiert werden, soll etwas Vernünftiges dabei herauskommen. Der Apple schlägt nur etwas schneller als unsere Ruderer: rund 1 Million mal in der Sekunde, oder, wie der Techniker sagt, mit 1 Megahertz (MHz).

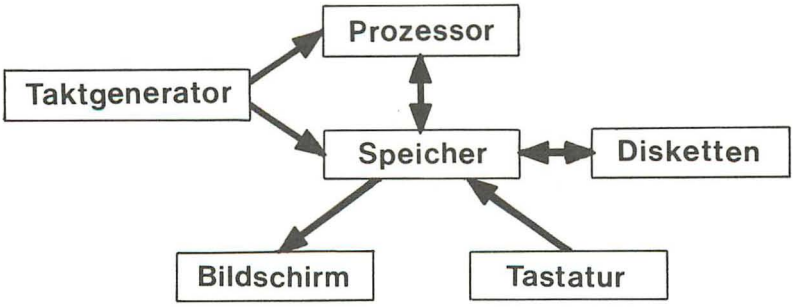


Abb. 1: Aufbau des Rechners

Im Anhang ist in einer Tabelle aufgeführt, wieviele dieser Takte der Prozessor für die Bearbeitung eines Befehls braucht (zwischen 2 und 7). Mit ihrer Hilfe können Sie sich selber leicht ausrechnen, wieviele Befehle der 6502 in der Sekunde verarbeitet. Dann wissen Sie auch, warum Maschinenprogramme so schnell sind.

Lektion 3: Wieviele Bits bitte?

Jedes technische Gerät erfordert, damit wir es beschreiben können, neue Wörter in unserer Sprache. Für den Umgang mit einem Computer müssen Sie Ihren Wortschatz etwas erweitern, wenn wir uns über die inneren Vorgänge unterhalten wollen. Außerdem müssen Sie zu Ihrem bisherigen 1x1 noch ein paar neue Regeln hinzulernen. Aber keine Angst, wir werden nicht sehr mathematisch, denn das Rechnen überlassen wir lieber dem Computer.

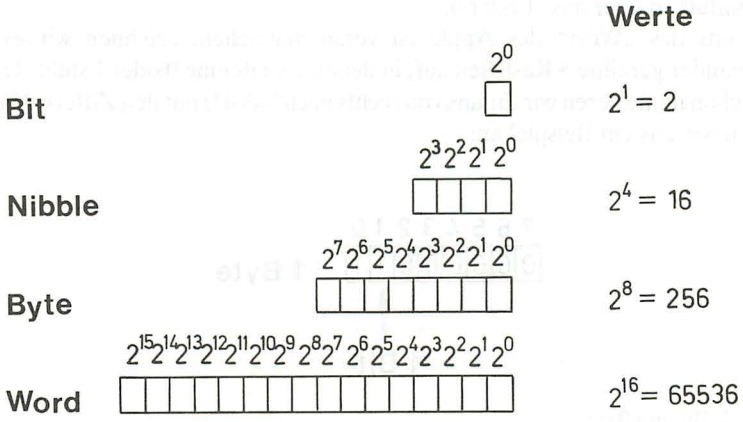


Abb. 2: Informationsmengen

Wie wir schon in Lektion 2 gehört haben, kennt der Rechner die logischen Zustände 0 und 1. Mit diesen beiden Möglichkeiten ist sein Ausdrucksvermögen aber sehr beschränkt. Würde der Rechner immer gleichzeitig zwei Leitungen bedienen könnnen, so wäre er damit bereits in der Lage, uns 4 verschiedene Informationen mitzuteilen: 00, 01, 10 und 11. Erhöhen wir die Zahl der Leitungen, so steigt der Informationsgehalt gewaltig an:

1 Leitung	2 Möglichkeiten
2 Leitungen	4 Möglichkeiten
3 Leitungen	8 Möglichkeiten
4 Leitungen	16 Möglichkeiten
5 Leitungen	32 Möglichkeiten

6 Leitungen	64 Möglichkeiten
7 Leitungen	128 Möglichkeiten
8 Leitungen	256 Möglichkeiten

Der Apple ist ein Rechner mit 8 Leitungen. Wir sagen daher, daß er eine Wortbreite von **8 Bit** oder **1 Byte** hat. Ein 16 Bit-Rechner hätte demnach die Wortbreite von 2 Byte oder 16 Leitungen.

Jedes Bit steht also für die Entscheidung zwischen zwei Dingen: Ja oder Nein, Stromfluß an oder aus, 1 oder 0.
Um uns das „Wort" des Apple zu veranschaulichen, zeichnen wir es als aneinander gereihte 8 Kästchen auf, in denen jeweils eine 0 oder 1 steht. Diese Kästchen numerieren wir für uns von rechts nach links(!) mit den Ziffern 0 bis 7. Sehen wir uns ein Beispiel an:

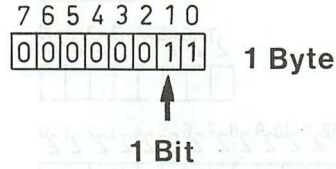


Abb. 3: Bit und Byte

Nun ist es etwas umständlich, immer Kästchen malen zu müssen. Lassen wir sie einfach weg, so erhalten wir
00000011.
Das ist eine sogenannte **Binärzahl** (Bin = Zwei), weil jede ihrer Ziffern nur zwei Zustände, eben 0 oder 1, annehmen kann (Dualsystem). Man kann mit ihnen ganz normal rechnen, z.B. addieren:

$$\begin{array}{r} 00000011 \\ +00000010 \\ \hline 00000101 \end{array}$$

Dabei gilt $0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 1$ und Übertrag zur nächsten Stelle, so wie Sie es gewohnt sind.

Eine Subtraktion verläuft ähnlich:

```
  00001001
- 00000111
----------
  00000010
```

Dabei gilt 0 - 0 = 0, 1 - 0 = 1, 1 - 1 = 0, 0 - 1 = 1 und Übertrag zur nächsten Stelle. Wenn Sie ein bißchen mit Binärzahlen üben wollen, starten Sie die Programme „**DUAL1**" und „**DUAL2**" von der Begleitdiskette. Diese führen Ihnen je 6 Beispiele vor. Danach können Sie selber Werte eingeben (immer 8-stellig) und sich ausrechnen lassen.

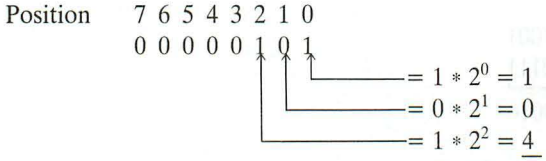
Was ein Byte (8 Bits) darstellt, hängt ganz davon ab, wie Sie es interpretieren. Zum Beispiel ist für den Computer das Byte 11000001 auf dem Bildschirm der Buchstabe „A". Als Hexadezimalzahl (mehr darüber im nächsten Kapitel) ist es $C1, dezimal 193 oder als vorzeichenbehaftete Ganzzahl (Integer) -63. Dasselbe Bitmuster kann ganz verschieden gedeutet werden. Wir werden uns all dieser Möglichkeiten noch annehmen.
Mit Binärzahlen zu rechnen ist nicht sehr schwierig, aber oft unpraktisch, da wir sehr schnell ganze Kolonnen von 0 und 1 benötigen, um große Zahlen darzustellen. Wir können nun Binärzahlen wieder in unsere gewöhnlichen Dezimalzahlen zurückverwandeln. Wie wir das „von Hand" tun, zeigt folgendes Beispiel.

Jede Position in einer Binärzahl bekommt einen Stellenwert zugewiesen. Dabei gilt:

Position	Wert
0	$1 = 2^0$
1	$2 = 2^1$
2	$4 = 2^2$
3	$8 = 2^3$
4	$16 = 2^4$
5	$32 = 2^5$
6	$64 = 2^6$
7	$128 = 2^7$

Wie Sie leicht sehen, ist jeder Wert doppelt so groß wie der davorstehende.

Zeichnen wir wieder ein Kästchenbyte



Um 00000101 in eine Dezimalzahl zu verwandeln, zählen wir die Positionswerte zusammen, an denen eine 1 steht.
Die größte mit 8 Bit darstellbare Zahl ist 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1= 255.
Um Binärzahlen eindeutig von Dezimalzahlen zu unterscheiden, stellen wir fortan allen Binärzahlen ein Prozentzeichen „%“ voran (es erinnert an 010!). %10 ist eine Binärzahl, 10 eine Dezimalzahl.

Lektion 4: HEX ist keine Hexerei

Wie Sie in Lektion 3 gesehen haben, können Sie mit einer 8-stelligen Binärzahl alle Dezimalzahlen zwischen 0 und 255 darstellen, insgesamt 256 Werte. Leider ist 255 für unser Gefühl ziemlich „krumm". Auf der anderen Seite sind Binärzahlen für die praktische Arbeit auch sehr unangebracht, da sie Fehler provozieren: wie leicht wird 100110111 mit 101100111 verwechselt!

Um all dem abzuhelfen, wurde ein neues Zahlensystem erfunden, das **Sedezimalsystem**. Dieses wird fälschlich oft auch **Hexadezimalsystem** genannt. Weil aber diese Fehler so verbreitet ist, werden auch wir ihn begehen und kurz von HEX-Zahlen sprechen. (Einem Gerücht zufolge soll 1963 bei IBM deshalb die Bezeichnung Hexadezimal- und nicht Sedezimal- oder Sexadezimalsystem gewählt worden sein, weil sonst die Abkürzung auf SEX-Zahlen hinausgelaufen wäre.)

Die Zahlenbasis eines Zahlensystems gibt an, wieviele verschiedene Zustände eine Ziffer annehmen kann. Im Dualsystem ist die Zahlenbasis 2, die möglichen Zustände sind 0 und 1. Im Dezimalsystem ist die Zahlenbasis 10, die möglichen Zustände sind 0 bis 9. Im Sedezimalsystem ist die Zahlenbasis 16 und die Zustände reichen von 0 bis F. Schauen wir uns eine Gegenüberstellung von Dezimal- und HEX-Zahlen an:

DEZ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Mit HEX-Zahlen lassen sich an einer Stelle 16 verschiedene Werte darstellen. Benötigen wir größere Werte, so stellen wir wie gewohnt eine zweite Ziffer davor. Nach F kommt 10, 11, 12, usw. bis 1F, dann folgen 20 bis 2F, 30 bis 3F usw. bis FF.
Noch größere Werte erfordern mehr Stellen. Im Anhang finden Sie eine Tabelle, um die HEX-Zahlen von 00 bis FF in Dezimalzahlen umzuwandeln. Die „Fragezeichen-Funktion" des Assemblers **„ASSESSOR"** (siehe dort) rechnet Ihnen bequem Zahlen bis zu 65535 zwischen allen drei Systemen hin- und her, inclusive Additionen und Subtraktionen zwischen Zahlen verschiedener Basis.

Sie können aber auch „von Hand" eine Konvertierung vollziehen. Wie schon bei den Binärzahlen müssen Sie dazu nur den Positionswert mit dem Ziffernwert multiplizieren und alles addieren. HEX-Zahlen haben folgende Positionswerte:

Position	Wert
0	$1 = 16^0$
1	$16 = 16^1$
2	$256 = 16^2$
3	$4096 = 16^3$

Mehr als 4 Positionen kommen im Zusammenhang mit dem Apple selten vor (z.B. bei ProDOS). Um eine HEX-Zahl deutlich von einer Dezimalzahl zu unterscheiden, stellen wir in diesem Buch fortan allen HEX-Zahlen ein Dollarzeichen „$" voran.

$10 ist eine HEX-Zahl,
%10 ist eine Binärzahl,
10 ist eine Dezimalzahl.

Wandeln wir nun $FF in eine Dezimalzahl um:
$FF = 15*16 + 15*1 = 255

Wir haben mit $FF das Äquivalent für die uns schon bekannte krumme Dezimalzahl 255. Daraus ist leicht ersichtlich, daß eine 8-stellige Binärzahl vollständig in einer nur 2-stelligen HEX-Zahl erfaßt werden kann.
%11111111 = $FF = 255

Schauen Sie sich die folgenden Beispiele an:
%00001111 = $0F
%11110000 = $F0

Sie ziehen den richtigen Schluß, wenn Sie feststellen, daß eine Stelle der HEX-Zahl in genau 4 Bit paßt und daß die linken vier Stellen von der linken HEX-Ziffer und die vier rechten Bits von der rechten HEX-Ziffer dargestellt werden. Weil sich unsere 8-stellige Binärzahl so schön symmetrisch zerlegt, wurden die Hälften auch noch benannt: die linke heißt das „höherwertige Nibble" und die rechte das „niederwertige Nibble". Allgemein gilt:

1 Byte = 2 Nibble = 8 Bit.

Auch HEX-Zahlen kann man addieren, subtrahieren, multiplizieren usw. Das müssen Sie jedoch nicht im Kopf können. Sie sollten lediglich im Laufe der Zeit ein gewisses Vorstellungsvermögen für HEX-Zahlen entwickeln und z.B. wissen, daß $E0 größer als $BF ist.

Wenn wir jetzt auf die Demonstrationen aus Lektion 1 zurückkommen, so werden Sie verstehen, warum wir im zweiten Beispiel von 0 bis 255 gezählt haben: hexadezimal ist das genau der Bereich von $00 bis $FF. Die Zahl 32767 ist schon etwas schwerer durchsichtig. Hexadezimal entspricht sie $7FFF. Zeichnen Sie diesen Wert doch einmal als Binärzahl mit 2*8 = 16 Stellen. Vielleicht fällt Ihnen dann etwas auf.

Lektion 5: Vorzeichen zum Vorzeigen

Mit einem Byte können wir nur Zahlen zwischen 0 und 255 darstellen. Was aber, wenn die Zahlen größer sind, oder wenn wir negative Zahlen benötigen? Das Problem mit den großen Zahlen ist noch verhältnismäßig einfach zu lösen: wir nehmen mehrere Bytes. Mit 2 Bytes kommen wir immerhin schon bis 65535 ($FFFF). In der hexadezimalen Schreibweise können wir die Aufteilung in 2 Bytes besonders leicht sehen: ein höherwertiges Byte (Hi) und ein niederwertiges Byte (Lo). In den meisten Fällen ist uns damit schon geholfen. Die Rechenoperation für solche 16-Bit Zahlen sind verhältnismäßig einfach und werden in späteren Kapiteln behandelt.
Ein größeres Problem sind negative Zahlen. Die gebräuchlichste Interpretation eines Bytes ist hier die vorzeichenbehaftete Ganzzahl (signed Integer). Eine Hälfte der 256 möglichen Bitmuster eines Bytes wird für positive Zahlen genommen, die andere für negative. Betrachten Sie einmal folgendes Bild:

```
127. 01111111
126. 01111110
125. 01111101
124. 01111100
       .
       .
       .

  4. 00000100
  3. 00000011
  2. 00000010
  1. 00000001
  0. 00000000
 -1. 11111111
 -2. 11111110
 -3. 11111101
 -4. 11111100
       .
       .
       .
```

-125. 10000011
-126. 10000010
-127. 10000001
-128. 10000000

Beachten Sie, daß alle Bitmuster, die mit einer 1 beginnen, als negativ interpretiert werden. Aus diesem Grund wird das Bit 7 (das Bit ganz links) auch als Vorzeichen-Bit (Sign-Bit) bezeichnet. Für den Computer ist Null (%00000000) damit eine positive Zahl, da Bit 7 nicht gesetzt ist.
Für alle Zahlen mit gelöschtem Bit 7 ist das Ergebnis der Interpretation als vorzeichenbehaftete Ganzzahl oder als normale Ganzzahl gleich: %01001111 ist immer $4F oder 79. Dagegen können wir %11111101 einmal als -3 oder aber als 253 auffassen. Achten Sie immer darauf, welche Interpretation gerade gefordert ist.
Wenn Sie nicht in einer Tabelle nachschlagen wollen, ist es praktisch, die negativen Zahlen berechnen zu können. Der einfachste Weg dazu ist, die positiv genommene Zahl von %11111111 zu subtrahieren und dann eine 1 zu addieren. Diese Form heißt „Zweier-Komplement" (Two's Complement), obwohl sie mit der Zahl 2 nichts zu tun hat. Sie addieren lediglich eine 1 zum Einer-Komplement der Zahl.

Beispiel -17:

```
  1 1 1 1 1 1 1 1
- 0 0 0 1 0 0 0 1
-----------------
  1 1 1 0 1 1 1 0   Einer-Komplement

+                 1
-----------------
  1 1 1 0 1 1 1 1   „Zweier-Komplement"
```

Dieses Verfahren ist besonders einfach, weil niemals ein Übertrag (Borgen) stattfinden muß, da immer von 1 subtrahiert wird.
Versuchen sie einmal selber, - 120 oder - 57 auszurechnen.

Lektion 6: Mit festem Wohnsitz

Nun werden Sie sich sicherlich fragen, ob sich die Mühe mit den HEX-Zahlen lohnt, nur um die Dezimalzahl 255 zu vermeiden. Bestimmt nicht nur deshalb, denn jetzt werden wir ein paar nützliche Anwendungen der HEX-Zahlen kennenlernen.
In Ihrem Apple wird nicht nur ein fester Takt geschlagen, hier regiert auch ein fast perfektes „Einwohnermeldesystem". Jedes Byte, das sich irgendwo im Rechner aufhält, hat eine Adresse. Der Hauptspeicher umfaßt $64 * 2^{10} = 65536$ Plätze, die je 1 Byte enthalten (Bei älteren Apple IIplus eventuell auch weniger). 64K (= Kilo-Byte) heißt das in einer mehr technischen Ausdrucksweise.

Wollen Sie in einer fremden Stadt ein bestimmtes Haus finden, benötigen Sie für eine eindeutige Identifikation den Namen der Straße und die Hausnummer. Beim Apple ist es ähnlich: Sie müssen die Speicherseite (Straße) und die Byteposition darin (Hausnummer) wissen. Der gesamte Speicher ist in 256 Seiten geteilt, die je 256 Byte enthalten ($256 * 256 = 65536$!). Die Seiten sind von $00 bis $FF numeriert, die Bytes darin ebenfalls von $00 bis $FF ($00 − $FF = 0 − 255 = 256 Zeichen, Sie erinnern sich?). So wie Sie Ihre Stadtadresse mit „Straße Hausnummer" angeben, genau so werden die Speicheradressen im Apple bezeichnet: erst die HEX-Zahl für die Seite, dann die HEX-Zahl für das individuelle Byte. Der Adressbereich reicht deshalb von $0000 bis $FFFF. Nun, sind Ihnen HEX-Zahlen schon sympathischer geworden?

Diese jetzt vierstellige HEX-Zahl setzt sich aus 2 Bytes zusammen: die beiden linken Zeichen stellen das höherwertige Byte (auch als Hi oder hi abgekürzt), die beiden rechten Ziffern das niederwertige Byte (als Lo oder lo abgekürzt) dar. Wir werden diese 2-Byte Adressen bei der Programmierung noch sehr häufig benutzen.

An bestimmten Speicherplätzen finden Sie immer die gleichen Informationen. Andere Speicher dienen stets den selben Funktionen und wiederum andere stehen für die freie Benutzung zur Verfügung. Die folgende Grafik zeigt uns den prinzipiellen Aufbau des Apple-Speichers, wobei wir einige Feinheiten zunächst noch unberücksichtigt lassen.

Seite		Adresse		Speichertyp	Funktion
255	$FF	65535	$FFFF	ROM	Monitor
248	$F8	63488	$F800		
247	$F7	63487	$F7FF	ROM	Applesoft-Interpreter
208	$D0	53248	$D000		
207	$CF	53427	$CFFF	—	Zusatz-ROMs (IIe/IIc) Ein- und Ausgabe sowie
192	$C0	49152	$C000		ROMs für Steckkarten
191	$BF	49151	$BFFF	RAM	freier Speicher
96	$60	24576	$6000		
95	$5F	24575	$5FFF	RAM	hochauflösende Grafik
64	$40	16384	$4000		HGR2
63	$3F	16383	$3FFF	RAM	hochauflösende Grafik
32	$20	8192	$2000		HGR1
31	$1F	8191	$1FFF	RAM	freier Speicher
8	$08	2048	$0800		
7	$07	2047	$07FF	RAM	Bildschirm für Text
4	$04	1024	$0400		oder Blockgrafik (GR)
3	$03	1023	$03FF	RAM	freier Speicher /
3	$03	768	$0300		Sprungvektoren
2	$02	767	$02FF	RAM	Tastaturpuffer
2	$02	512	$0200		
1	$01	511	$01FF	RAM	System-Stack (Stapel)
1	$01	256	$0100		
0	$00	255	$00FF	RAM	Zero-Page (Null-Seite)
0	$00	0	$0000		

Abb. 4: Speichereinteilung des Apple

Sie brauchen nicht sofort die ganze Speicherbelegung auswendig zu lernen. Im Laufe der Beschäftigung mit dem Apple wird ganz von selbst so etwas wie eine „geistige Landkarte" bei Ihnen entstehen.
Der gesamte Speicher des Apple ist noch einmal in zwei grundverschiedene Bereiche eingeteilt: Arbeitsspeicher und Festwertspeicher. Der Festwertspeicher enthält Daten und Programme, die nur gelesen, aber nicht geändert werden können. Man bezeichent den Festwertspeicher deshalb auch als Nur-Lese-Speicher oder ROM-Speicher, wobei ROM für „Read Only Memory" steht. Der ROM-Inhalt bleibt sogar nach Abschaltung des Stroms erhalten. Nur so ist es möglich, daß der Apple immer sofort arbeitsbereit ist.
Die Informationen im Arbeitsspeicher können sowohl von dort gelesen als auch dort hinein geschrieben werden. Man nennt diesen Speicher deshalb auch Schreib-Lese-Speicher oder RAM-Speicher, wobei RAM für „Random Access Memory" steht. Die Informationen im RAM bleiben nur solange erhalten, wie die Versorgungsspannung eingeschaltet ist. Wollen Sie Daten oder Programme aus dem RAM dauerhaft aufbewahren, so müssen Sie diese vor dem Abschalten des Apple auf ein externes Speichermedium wie eine Diskette oder Cassette überspielen.
Wir werden uns in den weiteren Lektionen mit allen Bereichen des Apple noch ausführlich befasssen, denn der Assembler-Programmierer ist selbst dafür verantwortlich, welche Speicherplätze er für seine Programme und Variablen verwendet.

Lektion 7: Ein Spaziergang im Monitor

Sie halten es wahrscheinlich für ganz selbstverständlich, daß Ihr Apple Sie nach dem Anschalten mit seinem Namen auf einem sonst gelöschten Bildschirm begrüßt und das Diskettenlaufwerk 1 anwirft. Verantwortlich hierfür ist der System-Monitor, ein 2 KByte langes Maschinenprogramm in den Speicherseiten $F8 bis $FF, Es gibt mittlerweile 5 verschiedene Versionen dieses Monitors: a) der Alte Monitor (Apple II), b) der Autostart-Monitor (Apple IIplus), c) der alte IIe-Monitor, d) der neue IIe-Monitor und e) der IIc-Monitor.
Wir wollen uns jetzt nicht näher mit den Unterschieden befassen, sondern mehr auf die Gemeinsamkeiten schauen. Alle fünf bestehen aus einer Vielzahl von kleinen Unterroutinen, die im Zusammenwirken für alle grundlegenden Operationen des Apple verantwortlich sind. Wir wollen nach und nach einige davon erforschen, da sie uns helfen, das Innenleben des Apple besser kennenzulernen. Zudem stellen sie einen Schatz von Funktionen dar, den wir quasi gratis für die eigene Programmierung mit benutzen dürfen.

Schalten Sie doch einmal den Apple mit der „System Master"-Diskette in Laufwerk 1 ein. Sobald das Bereitschaftszeichen (Prompt) erscheint (Ü oder]), tippen sie „CALL-151<RETURN>". Sie sollten dann ein Sternchen sehen. Dieses Sternchen ist das Bereitschaftszeichen des System-Monitors, der nun auf Ihre Eingaben wartet. Im Apple Benutzerhandbuch sind zahlreiche Befehle aufgeführt. Probieren wir einmal einige davon aus. Geben Sie „FC58G<RETURN>" ein. Der Bildschirm sollte gelöscht werden und das Sternchen links oben erscheinen. Was haben wir eigentlich mit unserem Befehl ausgelöst?

Als erstes müssen Sie wissen, daß Sie mit dem Monitor **nur** in HEX-Zahlen kommunizieren können, denen sie **kein** Dollarzeichen „$" voranstellen dürfen, da dezimale Eingaben ohnehin nicht erlaubt sind. Mit „FC58" haben Sie dem Monitor eine Adresse mitgeteilt, nämlich $FC58. Das „G" steht für „Go" (Gehen) und besagt, daß ab der angegebenen Adresse alle Befehle des dort stehenden Programms ausgeführt werden sollen. $FC58 liegt im System-Monitor selbst und ist die Startadresse der „HOME"-Routine. „FC58G<RETURN>" im Monitor bewirkt also dasselbe wie „HOME <RETURN>" in Applesoft. Es ist sogar dasselbe, denn der Applesoft-Interpreter ruft nach dem „HOME"-Befehl genau diese Monitor-Routine auf.

Geben Sie nun „FBDDG<RETURN>“ ein. Sie sollten einen Piepton hören. Sie haben die selbe Routine gestartet, die auch nach „<CTRL-G>“ in Applesoft den Lautsprecher betätigt. Wie wird nun dieser Ton erzeugt? Der Microprozessor führt ein Programm aus, daß in den Speicherstellen $FBDD bis $FBEF steht . Schauen wir uns dieses Programm einmal näher an. Geben Sie bitte „FBDD.FBEF<RETURN>“ ein, und Sie sollten folgendes Bild erhalten:

```
FBDD- A9 40 20
FBE0- A8 FC A0 C0 A9 0C 20 A8
FBE8- FC AD 30 C0 88 D0 F5 60
*
```

Sie haben mit dem obigen Befehl, der ganz allgemein „*Startadresse.Endadresse*<RETURN>“ lautet, einen Speicherauszug hergestellt (Memory Dump). Ganz links steht jeweils die vierstellige Adresse, dann ein Strich. Es folgen bis zu 8 HEX-Zahlen. Sie stellen den Code dar, den der Microprozessor abarbeitet, sie sind sein **Maschinenprogramm**. Unser 6502 (und 65C02) ist in der Lage, diese Befehle direkt auszuführen und etwas mehr oder weniger Sinnvolles zu tun. Für die meisten Menschen sind sie allerdings nur eine Ansammlung von unverständlichen Zahlen.

In den Entstehungstagen des Apple wurde programmiert, indem solche Ziffernfolgen in den Rechner eingetippt wurden. Da das auf die Dauer ein sehr mühseliges Geschäft war, ersann man eine Sprache, die dasselbe bewirkte, aber für einen Menschen besser begreifbar war, die **Assemblersprache**. Ihr Wortschatz ist nicht sehr groß: für den 6502/65C02 weit weniger als 100 „Wörter“ mit je 3 Buchstaben, den sogenannten „**Mnemonics**“ (sprich: ne mon iks, „Mnemonica“ ist die „Gedankenkunst“). Die Mnemonics sind abgeleitet von den englischen Beschreibungen dessen, was der Befehl im Prozessor bewirkt. „LDA“ z.B. steht für „LoaD Accumulator“, auf deutsch „LaDe den Akkumulator“. In Kapitel 4 finden Sie alle Mnemonics in alphabetischer Reihenfolge mit ausführlichen Erläuterungen. Diese dreibuchstabigen Kunstwörter werden von einem Programm, dem **Assembler**, in reine Maschinensprache verwandelt, bevor der Microprozessor sie ausführt.

Assembler
Assemblerprogramm <==========> Maschinenprogramm
Disassembler

Die Rückübersetzung ist auch möglich. Sie benötigen dazu einen Disassembler. Ein solcher ist bereits im System-Monitor enthalten und wird durch den Befehl „*Startadresse*L<RETURN>“ aktiviert (L = List).

Geben Sie bitte „FBDDL<RETURN>“ ein. Sie sollten folgendes Bild sehen (ohne die Kommentare):

```
*FBDDL

FBDD-   A9 40       LDA   #$40          Hier beginnt die Routine
FBDF-   20 A8 FC    JSR   $FCA8
FBE2-   A0 C0       LDY   #$C0
FBE4-   A9 0C       LDA   #$0C
FBE6-   20 A8 FC    JSR   $FCA8
FBE9-   AD 30 C0    LDA   $C030         Adresse des Lautsprechers
FBEC-   88          DEY
FBED-   D0 F5       BNE   $FBE4
FBEF-   60          RTS                 und hier endet sie.
FBF0-   A4 24       LDY   $24           Der folgende Code gehört
FBF2-   91 28       STA   ($28),Y       schon zur nächsten
FBF4-   E6 24       INC   $24           Routine „STOREADV"
FBF6-   A5 24       LDA   $24
FBF8-   C5 21       CMP   $21
FBFA-   B0 66       BCS   $FC62
FBFC-   60          RTS
FBFD-   C9 A0       CMP   #$A0
FBFF-   B0 EF       BCS   $FBF0
FC01-   A8          TAY
FC02-   10 EC       BPL   $FBF0
*
```

Der Monitor disassembliert immer 20 Zeilen, so daß Sie jetzt etwas mehr sehen als nur unsere Piep-Routine (BELL1, wenn Sie es lieber englisch mögen). Ganz links erscheint die Adresse, dann der dort stehende Maschinencode und schließlich rechts der entsprechende Assemblercode. Schon bald werden Sie seine Bedeutung entziffern können. Aber spielen wir vorerst noch etwas weiter. Der Monitor gestattet es Ihnen auch, einzelne Werte eines Maschinenprogramms zu verändern, indem Sie die gewünschte Adresse nennen, gefolgt von einem Doppelpunkt und dem neuen Wert.
Geben Sie jetzt bitte „FBE3:50<RETURN>“ ein und starten Sie danach die Routine mit „FBDDG<RETURN>“ erneut. Haben Sie einen Unterschied gehört? Wenn ja, müssen Sie einen ganz besonderen Apple besitzen. Geben Sie wieder „FBDDL<RETURN>“ ein. Sie sollten noch das Gleiche sehen wie im ersten Versuch. Der Monitor steht nämlich im ROM, und das können Sie nur lesen und ausführen, aber nicht verändern. Versuchen Sie es trotzdem, dann bekommen Sie *keine* Fehlermeldung, sondern der Apple ignoriert den Befehl einfach. In dieser Hinsicht ist die Maschinensprache nicht so freundlich wie etwa Applesoft, das Ihnen mit vielen Hinweisen (SYNTAX ERROR IN ...) unter die Arme greift. Wenn Sie in der Assemblersprache programmieren, sind Sie selbst

dafür verantwortlich, die Fehler zu suchen (aber auch dafür gibt es wieder Programme, die Debugger wie etwa „**IDUS**").

Wie uns dieses Beispiel gezeigt hat, können wir Monitor-Routinen zwar benutzen, aber wir müssen sie unverändert hinnehmen. Wenn sie aus irgend einem Grunde nicht ganz passen, bleibt uns nur, sie im RAM-Speicher nachzubilden und dort zu verändern oder gleich eine ganz eigene Routine zu schreiben. In wenigen Fällen ist es noch möglich, an einer anderen Stelle in eine Monitor-Routine einzuspringen und dadurch ein verändertes Ergebnis zu erzielen.

Zum Abschluß verschieben wir mit dem Monitor unsere Piep-Routine in den RAM-Speicher. Der Befehl dazu lautet:
„*Zieladresse* <*Startadresse.Endadresse*M<RETURN>" (M = Move). Dadurch wird der Bereich zwischen „Startadresse" und „Endadresse" in den Bereich ab „Zieladresse" kopiert.

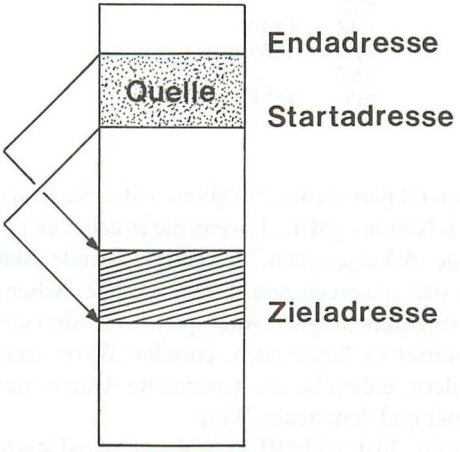


Abb. 5: Verschiebung im Speicher

Versuchen Sie es mit „1000<FBDD.FBEFM<RETURN>". Geben Sie jetzt „1000G<RETURN>" ein. Sie sollten keinen Unterschied zur normalen Routine feststellen können.
Nun versuchen Sie erst „1008:20<RETURN>" und dann „1000G <RETURN>". Jetzt sollte der Ton viel tiefer sein. Mit „1000L<RETURN>" können Sie sich Ihr Programm ansehen. Sie finden es von $1000 bis $1012. Alles danach ist Computermüll (engl.: Garbage).

Lektion 8: Vom Kopf zum Chip

Bevor wir jetzt tiefer in die Programmierung eindringen, müssen wir uns eines nicht ganz trivialen Problems annehmen: Wie kommt das Programm vom Kopf in den Rechner?

Um es nicht zu kompliziert werden zu lassen, nehmen wir ein einfaches Beispiel: Ein Computer-Benutzer hat die **Idee**, auf dem Bildschirm die Meldung „Guten Morgen!" erscheinen zu lassen. Um dies in die Realität umzusetzen, muß er sich zunächst einmal Gedanken über das **Problem** machen. Er wird feststellen, daß es nicht unlösbar ist: eine Zeichenkette (= String) soll auf dem Bildschirm ausgegeben werden. Jetzt muß er seine geistige Leistung fortsetzen und die Idee in ein ablauffähiges Programm umsetzen. In seinem Kopf (also immer noch in geistiger Form) entsteht ein **Plan**, der die Struktur des zukünftigen Programms beinhaltet. Ist unser Computer-Benutzer sehr sorgfältig, bringt er diesen Plan zu Papier (es gibt verschiedene geläufige Diagrammformen dazu).
Dann entscheidet er sich für eine **Computersprache**, in der er das Problem lösen will. Er wählt Assembler. Nach und nach entstehen in seinem Gehirn Folgen von Befehlen, die er in Tastendrucke umsetzt: das **Assemblerprogramm** wird eingetippt. Aus dem Code im Gehirn werden **Buchstaben- und Zahlentasten**, die betätigt werden. Die Tastatur des Rechners wandelt diese sofort wieder um in elektrische Signale, die zumeist dem **ASCII-Code** folgen (siehe Lektion 10).
Im Rechner läuft parallel dazu ein Hilfsprogramm, daß die Zeichen einsammelt und „vernünftig" ablegt: ein Editor wie z.B. der ASSESSOR-Editor. In den Speicherstellen des Rechners entsteht eine Sammlung von ASCII-codierten Informationen, der **Assembler-Quellcode**. Dieser ist für den Microprozessor noch gänzlich unverständlich. Wenn der Quellcode komplett eingegeben ist, wandelt deshalb ein weiteres Programm, ein Assembler wie der ASSESSOR-Assembler, diesen Quellcode um in **Maschinensprache (Objektcode)** und speichert ihn an einer anderen Stelle im Rechner.
Diese Umwandlung ist ein komplizierter Übersetzungsvorgang, der viele Regeln beachten muß. Der große Quellcode aus Buchstaben und Zahlen wird zu einem kompakten Maschinencode, der nur noch aus Zahlencodes besteht. Dieser Maschinencode ist es endlich, der vom Microprozessor verstanden wird, und wenn der Kopf unseres Benutzers gut gearbeitet hat, dann sollte auch tatsächlich „Guten Morgen!" auf dem Bildschirm erscheinen.

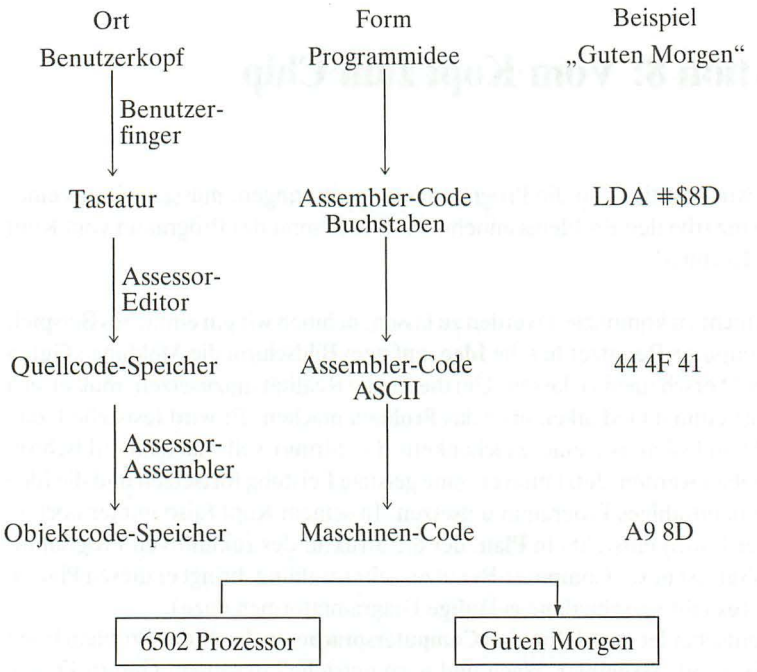


Abb. 6: Vom Kopf zum Chip

Lektion 9: Legale Tauschgeschäfte

Die Aufgabe des Microprozessors ist es, in vielen kleinen Schritten ein Maschinenprogramm zu decodieren und auszuführen. Dabei werden zum einen Teil die Inhalte von Speicherstellen verändert, zum anderen Teil finden die Operationen nur innerhalb des Microprozessors selbst statt. Werfen wir jetzt einen Blick auf den 6502 aus der Sicht des Assemblerprogrammierers (ein Halbleiter-Ingenieur würde wahrscheinlich andere Dinge interessant finden).
Der 6502 (und auch der 65C02, den wir im folgenden Text nur noch dann extra erwähnen, wenn bei ihm Abweichungen auftreten) besitzt Speicherstellen, die je ein Byte laden, speichern oder anderweitig manipulieren können.

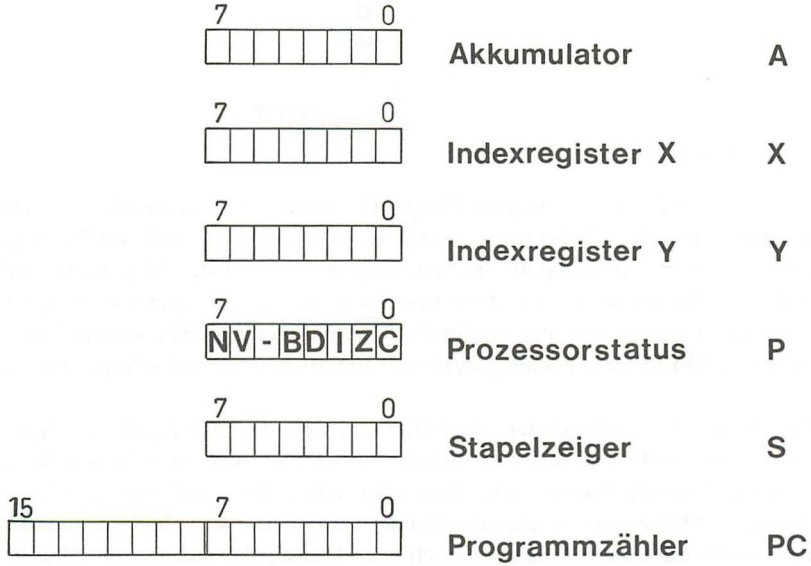


Abb. 7: Die Register des 6502/65C02

Der **Akkumulator** ist die wichtigste dieser Speicherstellen. Durch ihn hindurch verlaufen fast alle Operationen des Microprozessors. Ihm zur Seite stehen zwei weitere Speicher, die **X-** und **Y-Register** genannt werden. Wegen einer ihrer wichtigsten Funktionen heißen sie zusammen auch **Index-Register**. Der innnere Zustand des Prozessors wird durch 7 **Flaggen** oder Bedingungsanzeiger charak-

terisiert, die sich im **P-Register** befinden. 7 der 8 Bits dort stellen Flaggen dar, 1 Bit ist unbenutzt. Dieses Register heißt auch **Status-** oder **Prozessorstatus-Register**.
Ein weiteres 8-Bit Register ist der **Stackpointer** (Stapelzeiger), auch S-Register genannt. Er weist auf eine Speicherstelle innerhalb der Seite $01 (Adresse von $0100 bis $01FF) und wird z.B. für Unterroutinen benutzt. Wir können ihn auch als 2-Byte Adressen-Register verstehen, dessen höherwertiges Byte immer gleich $01 ist und dessen niederwertiges Byte sich im S-Register befindet.

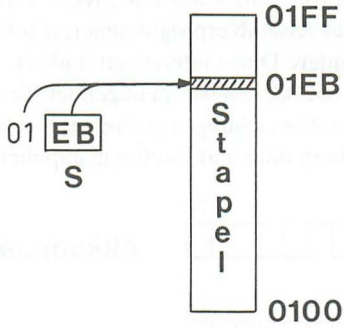


Abb. 8: Stapelzeiger

Schließlich finden wir noch einen **Program Counter** (Programmzähler, Adressregister), ein echtes 2-Byte bzw. 16-Bit Register. Es wird auch mit PC abgekürzt. Es zeigt auf die Stelle im Speicher, die als nächstes ausgeführt wird. Außer bei Verzweigungen und Sprüngen wird das Adressregister im Verlaufe einer Programmausführung kontinuierlich weitergezählt. Mit seinen 2 Byte ($0000 bis $FFFF) kann es auf jede der 65536 Speicherstellen des Apple weisen.

Zwischen den verschiedenen Registern und dem Hauptspeicher des Apple können auf vielfältige Weise die Inhalte verschoben werden. Die Abbildung zeigt mit ihren Pfeilen die möglichen Informationsflüsse auf. An den Pfeilen stehen die Mnemonics für die betreffende Aktion. Alle Verschiebungen *innerhalb* des Prozessors werden als Tausch oder Transfer bezeichnet (die Mnemonics beginnen mit „T"). Alle Verschiebungen zwischen dem Prozessor und dem Hauptspeicher sind Lade- und Speichervorgänge (die Mnemonics beginnen mit „L" oder „S").

Wie Sie sehen, sind nicht alle denkbaren Wege realisiert. Um z.B. den Stackpointer in einen Speicherplatz zu schreiben, müssen Sie ihn erst in das X-Register transferieren („TSX") und von dort in den Hauptspeicher schreiben

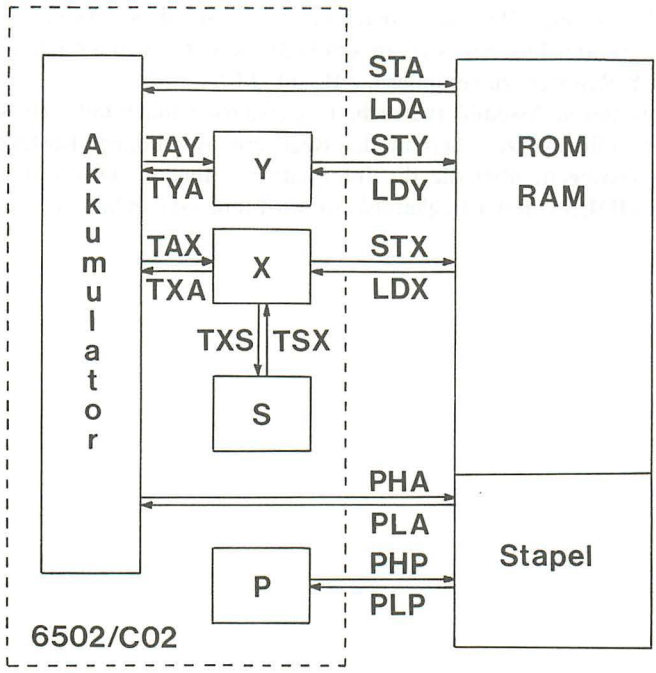


Abb. 9: Speicher- und Ladebefehle

(„STX"). Wir werden jetzt bald die einzelnen Schritte nachvollziehen. Schalten Sie jetzt bitte wieder Ihren Apple ein und gehen Sie in den Monitor (Sie erinnern sich an „CALL-151"?).
Tippen Sie bitte „<CTRL-E><RETURN>". <CTRL-E> erreichen Sie wie gewohnt, indem Sie die Control-Taste und den Buchstaben „E" gleichzeitig drücken.

```
:
A = FE   X = 01    Y = 00   P = 30   S = F8
```

<CTRL-E> ist der Monitor-Befehl, um die augenblicklichen Prozessorinhalte auszudrucken. A steht für „Akkumulator", X für „X-Register", Y für „Y-Register", P für „Prozessorstatus" und schließlich S für „Stapelzeiger". Die bei Ihnen angezeigten Werte können natürlich andere sein.
Geben Sie jetzt „:11 22 33 44 55 <RETURN>" ein. Bitte beachten Sie, daß nach dem Doppelpunkt **kein** Leerzeichen folgt, aber zwischen jedem Ziffernpaar 1 Leerzeichen stehen **muß**. Geben Sie wieder „<CTRL-E>

<RETURN>“ ein. Wie Sie sehen, haben Sie auf diese Weise direkt die Registerwerte des Prozessors verändern können. Versuchen Sie jetzt, nur den Wert des Y-Registers zu verändern, z.B. auf „FF“.
Für Ihre weiteren Assemblerversuche ist es wahrscheinlich hilfreich, sich den Prozessor bildlich vorzustellen mit den Registern als richtigen „Kästchen“ und Verbindungswegen, über die die Informationen fließen. Das umfangreiche Programm **IDUS** von der Begleitdiskette wird Ihnen dabei helfen.

Lektion 10: IDUS – So tun als ob

IDUS ist die Abkürzung für „Interaktiver Debugger Und Simulator". Dieses sehr nützliche und anschauliche Programm soll Ihnen ermöglichen, den Ablauf von Programmen (eigenen oder fremden) zu verstehen (Simulator), und es soll Ihnen bei der Fehlersuche behilflich sein (Debugger = „Entwanzer", da Fehler in der Computersprache als „Bugs" bezeichnet werden). IDUS reagiert während des Ablaufs auf eine Vielzahl von Befehlen, um Ihnen Steuerungsmöglichkeiten zu gewähren. Im Laufe dieses Kurses werden Sie seine Möglichkeiten schätzen lernen. Eine vollständige Bedienungsanleitung finden Sie in Kapitel 6 dieses Bandes.

Wir werden jetzt erste Gehversuche mit IDUS machen. Legen Sie die Begleitdiskette (die Kopie, Sie wissen doch!) ein und starten Sie das Programm mit „BRUN IDUS<RETURN>". Zunächst erscheint ein Titelbild, das einige Zeit stehen bleibt. Wenn Sie eine Taste drücken, verschwindet es sofort. Sie sehen dann eine Grafik, die die verschiedenen Teile des Microprozessors und des Speichers wiedergibt. Sie werden sicherlich die in Lektion 8 besprochenen Register des 6502 wiedererkennen. X- und Y-Register sind sowohl als Binärzahl als auch als HEX-Zahl dargestellt, der Akkumulator zusätzlich noch als ASCII-Zeichen (Control-Zeichen, z.B. $00, sind unsichtbar).
ASCII steht für „American Standard Code for Information Interchange" (= Amerikanischer Standard-Code für Informationsaustausch). Dieser genormte Code dient dem Apple und vielen anderen Rechnern zur Zeichendarstellung. Jedes Zeichen wird in einem Byte codiert, wobei nur die Bits 0 bis 6 benutzt werden. Damit lassen sich insgesamt 128 Zeichen (0 bis 127) wie Groß- und Kleinbuchstaben, Ziffern, Sonder- und Steuerzeichen definieren (Tabelle im Anhang). Das 8. Bit (Bit 7) wird oft rechnerintern als Unterscheidungsmerkmal herangezogen, ansonsten ist die zweite Hälfte (128 bis 255) identisch mit der ersten.

Die drei großen Rechtecke geben Ihnen Einblick in Teile des Apple-Speichers. Wir werden ihre Benutzung noch kennenlernen.

In der Mitte unten sehen Sie ein Menüfenster in inverser Schrift. Mit den Rechts- und Linkspfeiltasten können Sie neue Wahlmöglichkeiten durch dieses Fenster rollen (Rollmenü). Versuchen Sie es einmal. Beide Bewegungsrich-

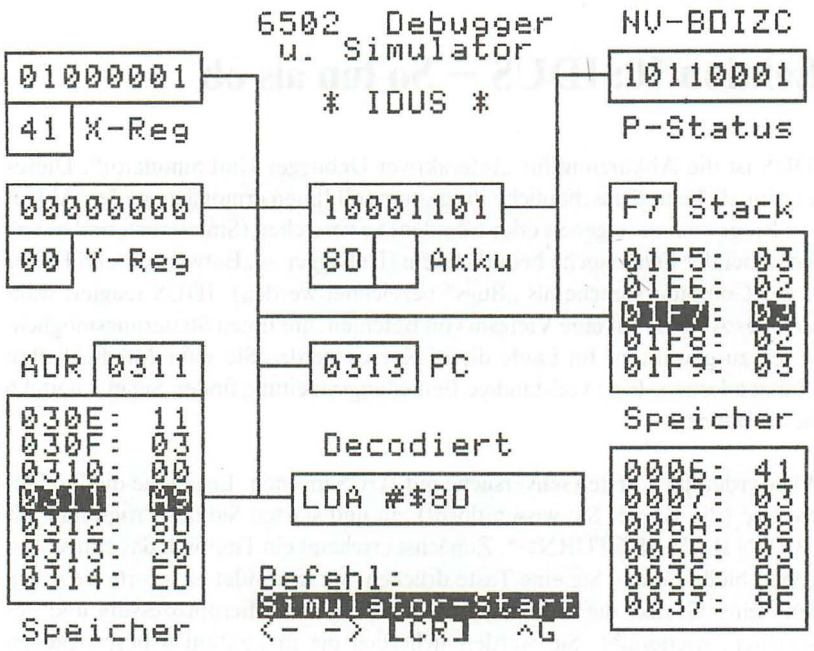


Abb. 10: Idus

tungen stoßen an kein Ende, sondern beginnen einfach von vorne. Steht die gewünschte Option in der Anzeige, brauchen Sie nur <RETURN> zu drücken. Eine Besonderheit ist der Befehl <CTRL-G>, der **immer** angenommen wird und die Bildschirmgrafik neu aufbaut. Dies ist für den Notfall gedacht, falls sich mysteriöse Dinge in der Grafik abspielen. Mehr darüber folgt später. Es ist aber ungefährlich, <CTRL-G> einzugeben, da dadurch keine gespeicherten Informationen verlorengehen (Warmstart). Versuchen Sie es ruhig einmal.

Für den Anfang wollen wir uns jetzt ein Programm ansehen, das eigentlich nichts Vernünftiges tut, außer daß es viele Informationsflüsse zwischen den Teilen des Prozessors in Gang setzt. Bringen Sie im Rollmenü die Option „File laden" zur Anzeige und wählen Sie sie mit <RETURN> an. Geben Sie dann als Namen „DEMO1<RETURN>" ein. Dadurch wird „DEMO1" von der Begleitdiskette geladen (sie muß natürlich im Laufwerk sein, sonst erhalten Sie eine Fehlermeldung). Ist dies erfolgreich geschehen, zeigt Ihnen IDUS den Programmbeginn für kurze Zeit an, um Ihnen eine Hilfe zu geben, wo Sie den Simulator starten können. In diesem Fall sollte die Adresse $0300 lauten.

Wählen Sie (per Rollmenü) nun „Simulator starten" aus und geben als Startadresse 0300 ein (300 funktioniert auch). Jetzt wird in Zeitlupe das Programm DEMO1 ausgeführt, wobei Sie ständig über alle Veränderungen im Prozessor unterrichtet werden. Wollen Sie den Ablauf zeitweilig stoppen, drücken Sie die <Leertaste>. Jede weitere Betätigung der <Leertaste> führt einen Programm(teil)schritt aus. Drücken Sie dagegen <RETURN>, setzt der „Dauerlauf" wieder ein. Mit <ESCAPE> können Sie die Simulation ganz verlassen und zum Rollmenü zurückkehren. Versuchen Sie es jetzt. Wählen Sie dann erneut „Simulator starten" an und geben Sie als Startadresse **nur** <RETURN> ein. Die Simulation wird dann dort fortgesetzt, wo sie mit <ESCAPE> abgebrochen wurde.

In dem großen Rechteck unten links (Speicher) sehen Sie wie durch eine Lupe einen kleinen Ausschnitt des Hauptspeichers, und zwar den Teil, in dem sich das ablaufende Programm DEMO1 befindet. In dem Feld „Decoder" wird jeder ausgeführte Maschinenbefehl, der ja nur als HEX-Zahl im Speicher steht, in Assemblersprache übersetzt. Verfolgen Sie eine zeitlang den Informationsfluß. Wenn es Ihnen zu schnell (oder zu langsam) geht, drücken Sie im Simulator die Rechts- oder Linkspfeiltaste. Sie sehen dann eine neue Anzeige im Menüfenster, die die augenblickliche Geschwindigkeit anzeigt. 00 ist die schnellste Gangart, FF die langsamste. Der Linkspfeil erniedrigt den Wert, der Rechtspfeil erhöht ihn. An den Skalenenden springt der Zähler um. Wird die gewünschte Geschwindigkeit angezeigt, setzt <RETURN> die Simulation wieder in Gang.

Wenn Sie wollen, können Sie die ausgeführten Assemblerbefehle (im Decoder angezeigt) in diesem Buch nachsehen und dann schauen, was im Prozessor geschieht. Gehen Sie in den Einzelschritt-Modus (<Leertaste>). Beim ersten Drücken wird dann nur der Befehl im Decoder disassembliert, erst beim zweiten Drücken wird er verarbeitet, so daß Sie Zeit haben, ihn nachzuschlagen. (Die Geschwindigkeit darf nicht auf 00 stehen, dann gibt es nur einen Schritt: Decodieren und Ausführen gleichzeitig).

Lektion 11: Nehmen, Geben und Tauschen

Der 6502 hat sechs Befehle, um seine Register zu laden oder zu speichern. Sie gehören zu den am meisten benutzten, und deshalb werden wir uns auch zuerst mit ihnen befassen.

Von BASIC oder anderen Sprachen kennen Sie Befehle wie „A = 10" oder „B = A". Dadurch erhält z.B. die Variable „A" den Wert 10 zugewiesen bzw. wird der Variablen „B" der Wert von „A" zugewiesen. In Assembler gibt es solche, nur mit einem Buchstaben bezeichneten Variablen nicht. BASIC sorgt intern dafür, daß „A" irgendwo im Speicher angelegt wird und danach mit seinem Namen angesprochen werden kann. In Assembler müssen Sie eine (oder mehrere) Speicherstellen definieren, die Sie für sich auch gern mit einem Variablennamen belegen dürfen. Den Microprozessor interessiert das nicht, er verarbeitet nur die absolute Adresse im Speicher.
Bleiben wir kurz bei „A = 10". Sie legen also z.B. fest, daß Ihr „A" in der Speicherstelle $1000 liegen soll. Dieses eine Byte kann einen Wert von 00 bis 255 ($00-$FF) annehmen, ist also ausreichend für die gewünschte „10". Um jetzt „10" nach „A" zu bringen, sind zwei Schritte notwendig:
a) Sie laden den Wert „10" in ein Register, zumeist den Akkumulator.
b) Sie speichern den Inhalt des Registers in „A", also $1000.
In einem Assemblerprogramm sieht das so aus:

```
LDA #10
STA $1000.
```

Mit dem X- oder Y-Register ergibt sich folgendes Bild:

```
LDX #10                    LDY #10
STX $1000                  STY $1000
```

Wir erkennen hier unsere dreibuchstabigen Mnemonics und dahinter einige Zahlenanweisungen. Das „LD" steht jeweils für „LoaD" (Lade), das „ST" für „STore" (Speichere). Der dritte Buchstabe legt das Register fest. Wenn Sie sich unsicher über eine Bedeutung sind, so schlagen Sie einfach im Befehlscode nach, der im Kapitel 4 ausführlich beschrieben ist.
Wir wollen eine Konstante (10) laden, d.h. wir wollen die „10" nicht aus irgendeiner Speicherstelle holen, sondern sie unmittelbar in das betreffende

Register übertragen. Dies machen wir dadurch kenntlich, daß wir vor die „10" ein Nummernkreuz „#" setzen. Wir haben damit ein Beispiel für die **„unmittelbare Adressierung"** (immediate), eine der 13 verschiedenen Adressierungsarten des 6502-Prozessors. Da „10" eine Dezimalzahl ist, braucht sie nicht weiter gekennzeichnet zu werden. Dann speichern wir das Register an der absoluten Adresse $1000 ab. Hier ist keine besondere Kennzeichung notwendig. Der Prozessor erkennt an der Adresse ($1000), daß die **„absolute Adressierung"** gemeint ist. Da es sich bei der Adresse um eine HEX-Zahl handelt, dürfen wir das „$" nicht vergessen.
Kommen wir zu „B = A". In BASIC verändert die Zuweisung „B = A" den Wert von „A" nicht, aber „B" erhält den selben Wert wie „A". In Assembler ist es ähnlich. Wenn wir ein Register mit einem Wert geladen haben und es dann irgendwo abspeichern, wird das Register nicht verändert. Wenn wir für uns festlegen, daß „B" gleich $1001 ist, so können wir folgenden Code schreiben:

```
LDA #10
STA $1000      ;A = 10
STA $1001      ;B = A
```

Mit dem X- oder Y-Register geht das natürlich genauso.

Nun möchten wir gerne noch, daß „C = $FA" wird. Wir schreiben dazu:

```
LDA #$FA
STA $1002      ;wenn wir C als $1002 festlegen
```

Zum Abschluß wollen wir, daß B und C ihre Werte tauschen. In BASIC müssen Sie dazu einen Ringtausch programmieren, damit keiner der Werte verloren geht: Zwischenspeicher = C, C = B, B = Zwischenspeicher. In Assembler können Sie einfach die verschiedenen Register benutzen, um einen Ringtausch zu organisieren:

```
LDA $1002      :C laden
LDX $1001      ;B laden
STA $1001      ;altes C nach B
STX $1002      ;altes B nach C
```

Jedes Register kann natürlich nur einen Wert enthalten, den es aber auch behält, wenn Operationen mit den anderen Registern ablaufen. Schreiben Sie dagegen

```
LDA #$FA
LDA #$0F
```

so wird $FA im Akkumulator sofort wieder durch $0F überschrieben .

Nun wollen wir einmal sehen, wie das Programm in der Praxis aussieht. Bisher steht es ja nur auf dem Papier und in unserem Kopf. Laden Sie wieder die Begleitdiskette und geben Sie „RUN START ASSESSOR<RETURN>“ ein. Nach kurzer Zeit begrüßt Sie der Editor und Assembler **ASSESSOR**. Eine vollständige Bedienungsanleitung finden Sie in Kapitel 5. Wir wollen uns jetzt nur mit ein paar Grundfunktionen beschäftigen, ohne gleich alle Möglichkeiten von ASSESSOR auszunutzen.
Um unser Programm einzugeben, rufen wir zunächst den Editor auf. Wenn Sie „&A<RETURN>“ eingeben, sollte auf dem Bildschirm eine „1“ erscheinen und dahinter der Cursor blinken. ASSESSOR hat eine automatische Zeilennumerierung und bei der Programmerstellung mehr Komfort als Applesoft.
Wir müssen jetzt zunächst ein paar Dinge tun, um den Assembler zufrieden zu stellen. Als erstes erwartet er ein Kommando, das ihm sagt, wo im Speicher das zu schreibende Programm einmal stehen soll. Wir wählen $0300, da hier ein schöner kleiner Platz ist für unser erstes Beispiel. Geben Sie bitte 1 Leerzeichen ein (wichtig!), dann in Großbuchstaben „ORG“, wieder ein Leerzeichen gefolgt von „$0300“. „ORG“ steht für „ORiGin“ (= Ursprung) und wird nicht in die Maschinensprache übersetzt, da es lediglich eine Anweisung für den Assembler darstellt, nicht für den Prozessor. Diese besondere Art von Mnemonics nennen wir auch „Pseudo-Opcodes“. Sie sind nicht genormt und können daher von Assembler zu Assembler verschieden sein.

Wenn Sie sich irgendwo verschreiben, können Sie mit den Pfeiltasten zurückgehen und den Fehler überschreiben. ASSESSOR besitzt einen vollständigen Zeileneditor mit Einfügen, Löschen, schnellen Cursorbewegungen usw. Sie sollten bald einmal die Bedienungsanleitung lesen. Schließen Sie Ihre erste Zeile mit <RETURN> ab. Automatisch erscheint die nächste Zeilennummer. Geben Sie jetzt folgendes Programm ein, wobei jede Zeile mit einem Leerzeichen beginnen muß und mit <RETURN> abgeschlossen wird.

```
LDA #10
STA $1000
STA $1001
LDA #$FA
STA $1002
LDA $1001
```

```
LDX $1002
STA $1002
STX $1001
RTS
```

Wenn Sie ein Leerzeichen vergessen oder ein Mnemonic falsch schreiben, bekommen Sie von ASSESSOR nach dem <RETURN> ein „? SYNTAX ERROR“ zu sehen und keine neue Zeilennummer. Geben Sie einfach wieder „&A<RETURN>“ ein, und die letzte Zeile kann erneut geschrieben werden. ASSESSOR hat sich die Zeilennummmer gemerkt. Das Mnemonic „RTS“ in der letzten Zeile besagt, daß hier das Programm zu Ende ist. Es steht für „ReTurn from Subroutine“ (Kehre zurück vom Unterprogramm) und sorgt dafür, daß Sie zu dem Programm zurückkehren, von dem diese Routine gestartet wurde. Das kann der Monitor sein, Applesoft, aber auch der Simulator IDUS. „RTS“ ist vergleichbar mit einem „RETURN“ in Applesoft.

Wenn Sie alle Zeilen eingegeben haben, drücken Sie beim Erscheinen der nächsten Zeilennummer nur „<RETURN>“. Damit wird der Editor verlassen. Um sich Ihr Werk anzusehen, geben Sie nun „&L<RETURN>“ ein, und Ihr Programm sollte schön tabuliert auf dem Bildschirm erscheinen.

Stellen Sie jetzt noch einen Fehler fest, so rufen Sie die Zeile mit „&*Zeilennummer*<RETURN>“ auf. Sie können dann innerhalb der Zeile korrigieren. Sind Sie damit fertig, geben Sie wieder „<RETURN>“ ein. Es kommt nicht darauf an, wo der Cursor dabei gerade steht. Im Gegensatz zu Applesoft wird immer die ganze Zeile bis zum Ende übernommen. ASSESSOR listet nun die nächste Zeile zur Korrektur. Wenn Sie diese überspringen wollen, geben Sie nur „<RETURN>“ ein und sofort erscheint die übernächste Zeile. Wollen Sie keine Korrekturen mehr machen, können Sie mit „<CTRL-X>“ ganz den Korrektur-Modus verlassen.

Jetzt ist der Quellcode fertig und wir können ihn in die Maschinensprache übersetzen. Geben Sie bitte „&ASM<RETURN>“ ein. Der Assembler meldet sich, und schon nach wenigen Sekunden sollte ein komplettes Assemblerlisting über ihren Bildschirm gegangen sein.

```
1. DURCHLAUF
 ..........

2. DURCHLAUF
                    1               ORG $300
0300: A9 0A         2               LDA #10
0302: 8D 00 10      3               STA $1000
0305: 8D 01 10      4               STA $1001
0308: A9 FA         5               LDA #$FA
030A: 8D 02 10      6               STA $1002
030D: AD 01 10      7               LDA $1001
0310: AE 02 10      8               LDX $1002
0313: 8D 02 10      9               STA $1002
0316: 8E 01 10      10              STX $1001
0319: 60            11              RTS

** ENDE **
```

Bekommen Sie statt der Meldung „** ENDE **“ eine Fehleranzeige, so korrigieren Sie den Quellcode bitte mit dem Editor und versuchen es erneut. Um das Ergebnis dauerhaft zu sichern, müssen Sie es auf Diskette abspeichern. Wenn wir es „MEIN PROGRAMM“ nennen, so lautet der Befehl dazu „&S“MEIN PROGRAMM““. Denken Sie an die Anführungszeichen vor und hinter dem Namen. Auf der Begleitdiskette ist so gut wie kein Platz mehr. Legen Sie sich für alle weiteren Programme eine eigene Diskette für ASSESSOR an, so wie es in Kapitel 5 beschrieben ist.
Nun wollen Sie sicher sehen, ob Ihr Programm auch funktioniert. Benutzen Sie dazu am besten den Simulator IDUS. Starten Sie ihn und laden Sie aus dem Rollmenü „MEIN PROGRAMM.OBJ“. Alles weitere läuft so wie gehabt. Einen Haken hat die Sache noch: woher sollen Sie wissen, daß die richtigen Werte auch bei $1000 usw. ankommen? Nun, hier hilft Ihnen die freie Speicheranzeige von IDUS. Wählen Sie im Rollmenü „Speicheranzeige“ und geben Sie dann „1000“ usw. ein. Maximal 6 Adressen sind möglich. Da Sie jetzt nur 3 benötigen, können Sie die Eingabe der letzten 3 überspringen, wenn Sie <ESCAPE> statt einer neuen Adresse eingeben. Jetzt können Sie sehen, ob alles an den richtigen Stellen ankommt.

Noch eine Anmerkung: Programme in Maschinensprache enden meist mit dem Befehl „RTS“. Wenn Sie ein solches Programm aus der Monitorebene starten, z.B. mit „300G<RETURN>“, stellt es für den Monitor praktisch eine Unterroutine dar, so daß der RTS-Befehl wieder in den Monitor zurückführt und die Programmausführung anhält. In den meisten Simulatoren wird durch RTS der Ablauf **nicht** beendet, sondern an völlig unbestimmbarer Stelle fortgesetzt, was

über kurz oder lang zumeist zum „Absturz" führt. IDUS hat Vorkehrungen getroffen, Sie beim abschließenden RTS in die Menüebene zurückzuführen. Dieser softwaremäßige Schutz kann jedoch manchmal, insbesondere bei fehlerhaften Programmen überlaufen werden. Drücken Sie dann die <ESCAPE>-Taste, eventuell gefolgt von <CTRL-G>. Hilft auch das nichts, bleiben nur die Notbremsen: <RESET> , Rechner ausschalten bzw. beim IIe und IIc der schonendere Neustart mit „<Offener Apfel, CTRL-RESET>".
Wenn Sie sich ein Programm ansehen wollen, setzen Sie *vor* dem Simulatorstart aus dem Rollmenü heraus erst immer den Stackpointer (Stapelzeiger) auf FD. Beim Kaltstart von IDUS erfolgt dies automatisch.

Lektion 12: Auf und Ab mit Rundenzähler

Wahrscheinlich wollen Sie mit dem Apple noch etwas mehr machen als nur Variablen zuzuweisen und Bytes zu verschieben. Jeder Computer kann rechnen, aber nicht jeder beherrscht auch gleich die vier Grundrechenarten. Unser 6502-Prozessor ist über die 1. Schulklasse nicht hinausgekommen: er addiert, subtrahiert und wenn es sein muß kann er auch noch mit 2 malnehmen und durch 2 teilen, alles andere müssen Sie durch Programme erledigen.
Betrachten wir zunächst zwei besonders einfache Fälle: die Addition von 1 und die Subtraktion von 1. Ersteres wird auch Inkrementierung genannt, letzteres Dekrementierung. Der 6502 hat sechs prinzipielle Möglichkeiten dazu, der 65C02 noch zwei mehr.

	Speicher	X-Register	Y-Register	Akku
Inkrement	INC	INX	INY	INA (65C02)
Dekrement	DEC	DEX	DEX	DEA (65C02)

Die direkte Anwendung auf den Akkumulator ist nur beim 65C02 möglich!
Wenn wir in BASIC schreiben „B = B + 1“, so können wir in Assembler dies ganz einfach nachvollziehen (B sei die Speicherstelle $1001 wie im letzten Kapitel):

```
INC $1001
```

So einfach geht das. Ein weiteres BASIC-Beispiel: „A = B + 1“.
Auch hier können wir in Assembler dasselbe mit wenigen Befehlen erreichen:

```
LDY $1001   ; das ist „B"
INY         ; + 1
STY $1000   ; das ist „A"
```

Wir laden also ein Register mit „B“, inkrementieren es (plus 1) und speichern es in „A“.
Hinter „INY“ steht keine Adresse, weil in dem Mnemonic bereits alle Informationen enthalten sind (Inkrementiere das Y-Register). Dies ist ein Beispiel für die **„implizite Adressierung“** (implied). Wir haben schon weitere Beispiele dafür kennengelernt: „TAY“ besagt z.B., daß der Inhalt des Akkumulators in das Y-Register transferiert werden soll. Aber auch „RTS“ enthält alle benötigten Informationen.

Die Subtraktion „A = B - 1“ geht genauso einfach:

```
LDY $1001   ; das ist „B"
DEY         ; - 1
STY $1000   ; das ist „A"
```

Da die Rechenoperation im Register stattfindet, ist „B“ jeweils unverändert, was ja auch erwünscht war. Wenn Sie dagegen

```
DEC $1001   ; „B" - 1
LDA $1001
STA $1000   ; nach „A" übertragen
```

programmieren, ist auch „B“ um 1 kleiner geworden. Dies entspräche „B = B - 1 : A = B“ in BASIC.

Wenn Sie einen Wert um zwei erhöhen oder erniedrigen wollen, können Sie die gerade gelernten Befehle auch mehrmals nacheinander verwenden:

```
LDY $1001   ; „B"
INY         ; + 1
INY         ; + 1
STY $1000   ; „A"
```

Dies ist gleichbedeutend mit „A = B + 2“.
Drei, vier, fünf usw. ließen sich so auch realisieren, aber dafür sind die „richtigen“ Additions- und Subtraktionsbefehle besser geeignet.

Bisher haben wir so getan, als könnten wir jeden beliebigen Wert so einfach erhöhen oder erniedrigen. Die Befehle verändern immer nur ein Byte, das sich entweder in einem Register des 6502 oder im Speicher befindet. Nun wissen wir aber, daß ein Byte nur die Werte von $00 bis $FF (0 - 255) enthalten kann. Nehmen wir an, das X-Register hätte schon den Wert $FF. Als Binärzahl ist das %11111111. Wenn wir jetzt noch 1 dazuaddieren, „klappt“ das Byte um und wir erhalten %00000000, was $00 entspricht. Wenn Sie ein wenig mit Binärzahlen zu rechnen geübt haben, werden Sie sicherlich jetzt bemerken, daß dabei eigentlich ein Übertrag in die 9. Stelle stattfinden müßte. Da unser Byte aber nur 8 Positionen hat, geht dieser Übertrag verloren. Der Prozessor bewahrt ihn auch nicht an anderer Stelle auf.
Umgekehrt sieht es aus, wenn Sie von $00 eine 1 abziehen:
%00000000 - 1 = %11111111.
Hier findet das „Umklappen“ in anderer Richtung statt.

In einem Programm ist es oft wichtig zu wissen, ob der Bereich eines Bytes überschritten wurde. Zu diesem Zweck hat der Prozessor in seinem Inneren ein 8 Bit (1 Byte) großes Register, daß Auskunft über die Ergebnisse (Status) der letzten Operationen gibt. Es dient also nicht zum Laden oder Speichern von irgendwelchen Werten. (Fast) jedes Bit darin ist eine Flagge (ein Signal) für ein Ereignis. Der Begriff „Flagge" ist dabei natürlich nur bildlich zu verstehen: hat das Bit den Wert 1, bedeutet das eine gesetzte Flagge, ist das Bit gleich 0, so ist die Flagge gelöscht (eingeholt).

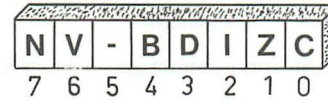


Abb. 11: Statusregister

Nun gibt es aber eine Flagge, die für uns jetzt interessant ist: Bit 1 ist die **Zero-** oder **Null-Flagge**. Sie wird gesetzt, wenn der Akkumulator, ein Indexregister oder eine Speicherstelle den Wert $00 als Ergebnis einer arithmetischen oder logischen Operation, eines Rotations-, Verschiebungs-, Inkrement- oder Dekrementbefehls oder durch Laden enthält. Sie wird gelöscht, wenn das Ergebnis von Null verschieden ist. Wenn wir z.B. zu $FE eine 1 mittels Inkrement addieren, wird die Flagge „0" sein, da das Ergebnis ($FF) nicht Null ist. Inkrementieren wir dagegen $FF noch einmal, wird das Ergebnis $00 und die Null-Flagge (Z) wird gesetzt.
Eine „1" zeigt also an, daß ein Zustand vorhanden ist, eine „0" zeigt, daß er nicht vorhanden ist. Bei der Null-Flagge mag es Sie vielleicht zunächst verwirren, daß das Ergebnis $00 mit einer „1" im Statusregister angezeigt wird. Betrachten Sie das Flaggen-Bit nicht als Zahl, sondern eben als Signal, das jetzt gesetzt ist.
Die Null-Flagge könnte uns anzeigen, daß ein Byte „umgeklappt" ist, wir also eine „Runde" hinter uns haben. Noch aber behält der Prozessor sein Geheimnis für sich. Wir müssen erst lernen, das Statusregister zu lesen.

Lektion 13: Ein Karussell mit Ausgang

Sicherlich haben Sie in BASIC schon oft den Befehl „IF ... THEN" benutzt. Hier wird eine Bedingung getestet, und wenn sie erfüllt ist, dann wird das Programm an der nach THEN stehenden Adresse fortgesetzt. In Assembler können wir etwas ganz ähnliches mit den „bedingten Verzweigungen" (Branches) machen. Der 6502 besitzt eine Reihe von Befehlen, die die Stellung einer Flagge im Statusregister testen und je nach Ergebnis entweder im Programm fortfahren oder aber zu einer anderen Stelle verzweigen, ganz wie in BASIC.
Mit dem Befehl „BEQ" (Verzweigen, wenn gesetzte Null-Flagge, Branch on EQual to zero) können wir immer einen Sprung ausführen, wenn das Ergebnis einer Operation Null war, mit „BNE" (Verzweige, wenn gelöschte Null-Flagge, Branch on Not Equal) testen wir genau die andere Bedingung.
Anders als BASIC besitzt ein Maschinenprogramm keine Zeilennummern. Wir müssen dem Prozessor deshalb die Programmspeicherstelle nennen, an der er fortfahren soll. Sehr schön wäre es, wenn wir ihm eine direkte Adresse sagen könnten. Die Verzweigungen in Assembler haben aber einen kleinen Haken: sie sind **relativ adressiert**. Anstatt dem Prozessor also im Klartext ein absolutes Sprungziel benennen zu können, müssen wir ihm eine relative Sprungweite angeben, die in nur einem Byte verschlüsselt ist (Distanz-Byte, Sprungweite, Displacement). Da die Sprünge sowohl vorwärts als auch rückwärts erfolgen sollen, muß das Distanz-Byte noch einmal aufgeteilt werden, so daß effektiv nur maximal 127 Bytes nach vorne und 128 Bytes zurück gesprungen werden kann. Die Distanz-Werte von $01 bis $7F gelten dabei als vorwärts, die Werte von $80 bis $FF als rückwärts, wobei $00 gar kein Sprung ist. Erinnert Sie das nicht an die vorzeichenbehafteten Ganzzahlen? Plus entspricht vorwärts, minus entspricht rückwärts.
Im Maschinencode steht das Distanzbyte direkt hinter dem Byte, das den Branch-Befehl darstellt. Die Befehlslänge ist damit 2 Bytes. Das Byte unmittelbar danach hat die relative Entfernung $00, das nächste $01 usw. In Abbildung 12 sehen Sie, wie das Distanzbyte bestimmt wird.

Rückwärts						Vorwärts		
*	*	*	*	B	D	*	*	*
^	^	^	^	^	^	^	^	^
FA	FB	FC	FD	FE	FF	00	01	02

* = Programm-Bytes
B = Branch-Befehl
D = Distanz-Byte

Abb. 12: relative Verzweigung

Wenn Sie mit einem Assembler wie z.B. ASSESSOR arbeiten, brauchen Sie die relativen Adressen nicht auszurechnen. Bei Ihrem Quellcode geben Sie ganz vorne in der ersten Spalte, wo Sie bisher immer ein Leerzeichen tasten mußten, eine Marke (Label) an. Diese etikettiert dann den Speicherplatz, an dem das Befehlsbyte steht. Der Assembler setzt später selber die echte Adresse dort ein und berechnet für Verzweigungen die Distanz-Bytes.
Programmieren wir mit den uns bisher bekannten Befehlen eine Warteschleife:

```
        LDY #$FF
SCHLEIFE DEY
        BNE SCHLEIFE
```

In der ersten Zeile initialisieren wir unser Zählregister auf $FF. Dann wird eine 1 subtrahiert. Das Ergebnis ist **nicht** Null, also ist die Bedingung „Not Equal" wahr und die Verzweigung führt zum Label „SCHLEIFE' zurück. Erst wenn auf $01 ein $00 folgt, ist die Bedingung nicht mehr erfüllt und die Schleife wird verlassen. In BASIC sähe das so aus:

```
10 A = 255
20 A = A - 1
30 IF A <> 0 THEN GOTO 20
```

Eine solche Schleife benötigt in Assembler natürlich nicht sehr lange. Wenn wir bis $FFFF zählen, ergibt das schon eine merkliche Verzögerung. Unser Zähler erfordert dann aber zwei Bytes, die wir als Hi-Zähler und Lo-Zähler irgendwo im Speicher definieren (Hi = High = höherwertiger Teil, Lo = Low = niederwertiger Teil). Wählen wir dafür z.B. die Speicherstellen $0400 und $0401. Wenn wir von $0000 bis $FFFF heraufzählen, muß der Lo-Zähler jeweils bis $FF inkrementiert und dann beim Schritt nach $00 (Z-Flagge gesetzt) auch der Hi-Zähler um 1 hochgesetzt werden.

```
        LDA #$00        ;Initialisierung
        STA $400        ;Lo-Zähler
        STA $401        ;Hi-Zähler
SCHLEIFE INC $400
        BNE SCHLEIFE    ;kein Übertrag
        INC $401
        BNE SCHLEIFE    ;noch nicht $FFFF
        RTS
```

Wenn auch der Hi-Zähler „überklappt" und damit Null wird, ist unsere Warteschleife beendet und das Programm geht geradeaus weiter. Hier trifft es auf den Befehl „RTS" (ReTurn from Subroutine, Rückkehr von der Unterroutine), der zum Aufrufer zurückführt, also in ein übergeordnetes Programm oder den Monitor oder nach Applesoft, je nachdem von wo aus Sie die Warteschleife aufgerufen haben.

Geben Sie zur Übung das Progamm in den Assembler ASSESSOR ein. Wählen Sie als „ORG" z.B. wieder $300. Wenn Sie beim Assemblieren keine Fehlermeldung erhalten, speichern Sie den Objektcode mit „&S"WARTEN"" auf Diskette ab. Mit „BRUN WARTEN.OBJ" können Sie es anschließend starten. Wir haben die beiden Zählbytes in den Bildspeicher gelegt, so daß Sie oben links ihre Veränderung sehen können. Wenn Sie jetzt enttäuscht sind, nicht die Zahlen 00 bis FF zu sehen, so bleibt nur die Erkenntnis: alles ist Ansichtssache. Die Bytes enthalten tatsächlich die Hex-Zahlen $00 bis $FF, aber Ihr Bildschirm interpretiert sie als Buchstaben und Zahlen in Normal, Inverse oder Flash. Die Bildschirmdarstellung des Apple ist eine eigene Lektion wert. Mit IDUS können Sie sich in aller Ruhe die Stellung der Flaggen (die Null-Flagge Z ist Bit 1 des Statusregisters) und auch unsere beiden Zählbytes ansehen, wenn Sie sich die Speicherstellen $400 und $401 im Fenster rechts unten anzeigen lassen.

Lektion 14: 1 + 1 = 3, auch das kann sein

Wir werden jetzt lernen, wie wir sowohl 8-Bit als auch 16-Bit Zahlen mit dem 6502 addieren. Besondere Bedeutung erlangt für uns der Akkumulator, den wir kurz Akku nennen. „Akkumulieren" heißt „ansammeln", und dieses Register ist so benannt, weil sich hier die Rechenergebnisse ansammeln. Alle Additionen (mit Ausnahme der um 1 mittels Inkrement) können nur im Akkumulator erfolgen.
Nehmen wir zunächst wieder ein kurzes BASIC-Beispiel: „A = B + C". Der normale Vorgang in Assembler sieht etwa folgendermaßen aus:

Lade B (in den Akkumulator)
Addiere C (so daß jetzt B + C im Akku ist)
Speichere die Summe (den Akku) in A.

Wenn unsere Zahlen größer als $FF werden, benötigen Sie mehr als je 1 Byte. In den meisten Fällen kommen wir mit zwei Bytes aus, also mit 16-Bit Zahlen. Dabei enthält ein Byte immer den höherwertigen Anteil (Hi) und das andere den niederwertigen Anteil (Lo). Unsere Addition wird jetzt etwas länger, da wir alle Operationen über den (einen) Akkumulator laufen lassen müssen:

Lade B(Lo) (in den Akkumulator)
Addiere C(Lo) (so daß jetzt die Summe im Akku ist)
Speichere die Summe (den Akku) in A(Lo)
Lade B(Hi) (in den Akkumulator)
Addiere C(Hi) (und eventuell eine weitere 1, wenn vorher ein
Übertrag entstanden ist)
Speichere die Summe in A(Hi).

Ein Übertrag wird beim 6502 im Statusregister vermerkt. Dazu dient ein Bit, die Übertrags- oder Carry-Flagge. Eine Addition mit Übertrag ist im 6502 als einzige Addition vorhanden, der Befehl lautet ADC (ADd with Carry, Addiere mit Übertrag). Wenn das Carry-Bit gesetzt ist (also „1" ist), wird es zusätzlich zum Akkumulator addiert: z.B. ergibt 1 + 1 + Carry dann eben 3. Ist das Carry-Bit gelöscht („0"), ändert sich nichts. Jeder ADC-Befehl beeinflußt das Carry-Bit erneut je nach seinem Ergebnis. War das Carry-Bit gesetzt und wird dann addiert, so kommt eine 1 zur Summe dazu. Liegt dieses neuerliche

Ergebnis dann unter $100 (also im Bereich bis $FF), wird die Carryflagge automatisch gelöscht. Die Carry-Flagge zeigt also immer die letzte Operation an.
Wenn wir bei unserer 16-Bit Addition die Lo-Bytes addieren, sollte natürlich noch kein Übertrag vorhanden sein. Der 6502 hat jedoch keinen Befehl für „Addiere ohne Übertrag". Wenn wir das Carry-Bit aber vorher auf Null setzen (löschen), kommt das im Ergebnis auf dasselbe hinaus. Der Befehl dazu lautet CLC (CLear Carry, Lösche die Carry-Flagge). Bei der Addition der Hi-Bytes wollen wir einen eventuellen Übertrag berücksichtigen. Wir lassen deshalb das Carry so, wie es durch die Addition der Lo-Bytes geworden ist: gelöscht, wenn das Ergebnis <= $FF war, gesetzt, wenn das Ergebnis > $FF war. Vollständig lautet unsere Addition also:

```
CLC         ; Carry löschen
LDA $302    ; „B(Lo)"
ADC $304    ; „C(Lo)"
STA $300    ; „A(Lo)"
LDA $303    ; „B(Hi)"
ADC $305    ; „C(Hi)"
STA $301    ; „A(Hi)"
```

Die Speicherstellen für unsere Variablen können Sie auch an anderem Ort stehen haben. Merken Sie sich für die Addition:

Erst Carry löschen (CLC), dann addieren (ADC).

Wenn wir zu einer Variablen eine Konstante addieren wollen, geht das genauso: „A = B + 10"

```
CLC         ; Carry löschen
LDA $1001   ; „B"
ADC #10     ; Konstante
STA $1000   ; „A"
```

Aus einer früheren Lektion kennen Sie ja schon die Vereinbarung, daß eine Konstante mit dem Nummernkreuz „#" gekennzeichnet wird. Auch drei 8-Bit Zahlen können Sie addieren:

```
CLC
LDA $1001
ADC $1002    ; A = B + C + D
ADC $1003
STA $1000
```

Nach jeder Addition, deren Ergebnis Sie nicht vorhersagen können, sollten Sie am Ende prüfen, ob nicht das Fassungsvermögen Ihrer Variablen überschritten wurde. Am Ende der Addition sollte das Carry-Bit immer gelöscht sein. Ist das nicht der Fall, war das Ergebnis zu groß. Mit BCS (Branch on Carry Set, Verzweige, wenn das Carry gesetzt ist) können wir diesen Zustand testen und im Fehlerfall zu einer entsprechenden Routine verzweigen:

```
CLC
LDA $1002
ADC #$55
STA $1002
BCS FEHLER
```

```
CLC
LDA $1002
ADC $1004
STA $1000
LDA $1003
ADC $1005
STA $1001
BCS FEHLER
```

Das linke Beispiel entspricht „B = B + Konstante“ in BASIC .

Lektion 15: Borgen bringt Sorgen

Zunächst einmal ist die Subtraktion von 8- oder 16-Bit Zahlen ganz ähnlich der Addition. Wenn wir A = B - C berechnen wollen, müssen wir folgende Schritte tun:

Lade B (in den Akkumulator)
Subtrahiere C (so daß die Differenz im Akku bleibt)
Speichere den Akku (die Differenz) in A.

Wenn unsere Zahlen größer als $FF werden, benötigen sie mehr als je 1 Byte. In den meisten Fällen kommen wir auch hier mit zwei Bytes aus, also mit 16-Bit Zahlen. Auch alle Operationen der Subtraktion müssen über den (einen) Akkumulator laufen:

Lade B(Lo) (in den Akkumulator)
Subtrahiere C(Lo) (so daß jetzt die Differenz im Akku steht)
Speichere die Differenz (den Akku) in A(Lo)
Lade B(Hi) (in den Akkumulator)
Subtrahiere C(Hi) (und eventuell eine weitere 1, wenn vorher ein
Borgen erforderlich war)
Speichere die Differenz in A(Hi).

Immer wenn wir eine größere Ziffer von einer kleineren abziehen, müssen wir von der nächsten Stelle eine 1 borgen. Wenn Sie ganz normal mit Papier und Bleistift zwei Zahlen subtrahieren, machen Sie es genauso. Nun hat der 6502 aber keine Flagge extra für ein Borgen, sondern er benutzt das Carry mit als Borge-Flagge.
Vor einer Subtraktion muß die Borge-Flagge mit „1" gefüllt werden, damit aus ihr auch etwas weggenommen werden kann. Da die Borge-Flagge mit dem Carry identisch ist, heißt das, daß das Carrybit gesetzt wird. Dazu gibt es den Befehl SEC (SEt Carry, Setze das Carrybit). Wenn die folgende Subtraktion mit SBC (SuBtract with Carry, Subtrahiere mit Übertrag) ein Ergebnis liefert, das größer oder gleich Null ist, ist kein Borgen erforderlich, die Borge-Flagge bleibt unangetastet auf „1". Liegt das Ergebnis dagegen unter Null, wird die „1" geborgt und das Carry damit auf „0" gesetzt. Bei der nachfolgenden Subtraktion ist das Borgen mit zu berücksichtigen, indem eine weitere 1 vom Ergebnis abgezogen wird.

Wenn das Carrybit gesetzt ist (also „1" ist), wird nichts zusätzlich vom Akkumulator subtrahiert, ist das Carrybit gelöscht („0"), wird eine 1 subtrahiert. Jetzt wird deutlich, warum die Borge-Flagge das Komplement zum Carry ist.
Jeder SBC-Befehl beeinflußt das Carrybit erneut je nach seinem Ergebnis.
Wenn wir bei unserer 16-Bit Subtraktion die Lo-Bytes subtrahieren, sollte natürlich noch kein Borgen vorher stattgefunden haben. Der 6502 hat aber keinen Befehl für „Subtrahiere ohne Übertrag". Wenn wir das Carrybit aber vorher mit SEC auf 1 setzen, kommt das im Ergebnis auf dasselbe hinaus. Bei der Subtraktion der Hi-Bytes wollen wir ein eventuelles Borgen berücksichtigen. Wir lassen deshalb das Carry so, wie es durch die Subtraktion der Lo-Bytes geworden ist: gesetzt wenn das Ergebnis >= $00 war, gelöscht, wenn das Ergebnis < $00 war. Vollständig lautet unsere Subtraktion also:

```
SEC         ; Carry setzen
LDA $302    ; „B(Lo)"
SBC $304    ; „C(Lo)"
STA $300    ; „A(Lo)"
LDA $303    ; „B(Hi)"
SBC $305    ; „C(Hi)"
STA $301    ; „A(Hi)"
```

Die Speicherstellen für unsere Variablen können Sie auch an anderem Ort stehen haben. Merken Sie sich für die Subtraktion:

Erst Carry setzen (SEC), dann subtrahieren (SBC).

Wenn wir von einer Variablen eine Konstante subtrahieren wollen, geht das genauso: „A = B - 10"

```
SEC         ; Carry setzen
LDA $1001   ; „B"
SBC #10     ; Konstante
STA $1000   ; „A"
```

Auch hier wird die Konstante mit dem Nummernkreuz „#" gekennzeichnet. Auch drei 8-Bit Zahlen können Sie subtrahieren

```
SEC
LDA $1001
SBC $1002    ; A = B - C - D
SBC $1003
STA $1000
```

Nach jeder Subtraktion, deren Ergebnis Sie nicht vorhersagen können, sollten Sie am Ende prüfen, ob nicht ein ungültiges Ergebnis entstanden ist. Am Ende der Subtraktion sollte das Carrybit immer gesetzt sein. Ist das nicht der Fall, war das Ergebnis unter Null. Mit BCC (Branch on Carry Clear, Verzweige, wenn das Carry gelöscht ist) können wir diesen Zustand testen und im Fehlerfall zu einer entsprechenden Routine verzweigen:

```
SEC
LDA $1002
SBC #$55
STA $1002
BCC FEHLER
```

```
SEC
LDA $1002
SBC $1004
STA $1000
LDA $1003
SBC $1005
STA $1001
BCC FEHLER
```

Das linke Beispiel entspricht „B = B - Konstante“ in BASIC .

Es ist ganz wichtig, daß Sie die Funktion des Carrybits im Zusammenhang mit Additionen und Subtraktionen verstehen. Der Befehl ADC addiert eine zusätzliche 1, wenn das Carrybit *gesetzt* (1) war, der Befehl SBC subtrahiert eine zusätzliche 1, wenn das Carrybit *gelöscht* (0) war. Anders ausgedrückt: ADC addiert den Inhalt des Carrybits, SBC subtrahiert das Komplement des Carrybits, da auch die Borge-Flagge das Komplement der Überlauf-Flagge ist. Inkrement und Dekrement beeinflussen das Carrybit **nicht**. Ebenso ohne Wirkung bleiben Lade- und Speicherbefehle wie LDA und STA. Die folgenden Beispiele sollen Ihnen diesen Tatbestand noch einmal verdeutlichen. Sie können diese oder andere Beispiele auch mit dem ASSESSOR assemblieren und sich mit IDUS die Flaggen ansehen. Das Carrybit ist mit „C“ im Prozessor-Status abgekürzt und steht ganz rechts im Byte.

```
CLC       ; C = 0
LDA #$FF  ; C = 0
ADC #$01  ; C = 1
  A = $00
```

```
SEC       ; C = 1
LDA #$00  ; C = 1
SBC #$01  ; C = 0
  A = $FF
```

```
CLC       ; C = 0
LDY #$FF  ; C = 0
INY       ; C = 0!
  Y = $00
```

```
SEC       ; C = 1
LDY #$00  ; C = 1
DEY       ; C = 1!
  Y = $FF
```

```
LDA #$FE  ; C = ?
CLC       ; C = 0
ADC #$01  ; C = 0
INA       ; C = 0
  A = $00
```

```
LDA #$01  ; C = ?
SEC       ; C = 1
SBC #$01  ; C = 1
DEA       ; C = 1    (65C02)
  A = $FF
```

```
CLC         ; C = 0
LDA #$FF    ; C = 0
ADC #$01    ; C = 1
SBC #$01    ; C = 0
  A = $FF
```

```
CLC         ; C = 0
LDA #$FF    ; C = 0
INA         ; C = 0   (65C02)
DEA         ; C = 0   (65C02)
  A = $FF
```

```
CLC         ; C = 0
LDA #$FF    ; C = 0
SBC #$01    ; C = 1
ADC #$01    ; C = 0
  A = $FF
```

```
CLC         ; C = 0
LDA #$FF    ; C = 0
DEA         ; C = 0   (65C02)
INA         ; C = 0   (65C02)
  A = $FF
```

Wenn Sie mit dem letzten Beispiel (links) Schwierigkeiten haben, sehen Sie es sich in Einzelschritten im Simulator an.

Lektion 16: Zeige Flagge bei Vergleichen

In Lektion 12 haben wir schon die Null-Flagge (Zero-Flag) benutzt um festzustellen, ob eine Operation das Ergebnis „Null" hatte. Mit den zwei bedingten Verzweigungen BEQ und BNE hatten wir die Stellung dieser Flagge im Prozessorstatus-Register getestet. BCC und BCS dienten uns in Lektion 14 und 15 dazu, einen Übertrag bzw. ein Borgen bei Addition und Subtraktion festzustellen. Wir wollen in dieser Lektion noch weitere Bedeutungen dieser Flaggen kennenlernen, die uns bei Vergleichen nützliche Informationen liefern.
In BASIC sind Ihnen sicherlich einfache Vergleiche geläufig:

```
IF A = B THEN GOTO ...
IF B <> C THEN GOSUB ...
IF D < 100 THEN ...
```

In Assembler besitzen wir dazu die drei Vergleichsbefehle

CMP	Vergleiche mit dem Akkumulator
CPX	Vergleiche mit dem X-Register
CPY	Vergleiche mit dem Y-Register.

Die ersten beiden BASIC-Beispiele lassen sich einfach umsetzen:

Lade A in den Akkumulator (oder in ein Indexregister)
Vergleiche das Register mit B
Verzweige bei Gleichheit (oder Ungleichheit)

Es sei A wieder $1000, B gleich $1001 und C die Speicherstelle $1002.

```
LDA $1000   ; A              LDA $1001   ; B
CMP $1001   ; B              CMP $1002   ; C
BEQ ...     ; gleich         BNE ...     ; ungleich
```

Jeder Vergleichsbefehl bewirkt im Prozessor eine Subtraktion (immer ohne Borgen unabhängig vom Carry-Bit), bei der die getestete Speicherstelle vom Inhalt des Registers abgezogen wird. Im ersten Fall wird also A − B gerechnet. Das Subtraktionsergebnis wird aber nicht aufbewahrt wie bei einer echten Subtraktion (SBC), sondern es setzt nur die Flaggen im Prozessorstatus entsprechend. Wenn A gleich B ist, dann ist A − B = 0. Deshalb wird bei Gleichheit die Null-Flagge (Zero-Flag) gesetzt. Wenn A und B verschieden sind, ist A − B <>

0 und die Null-Flagge wird gelöscht. Da die beiden Befehle BEQ und BNE die Stellung der Null-Flagge testen, können sie zur Entscheidungsfindung dienen. BEQ (Branch on EQual = Verzweige falls gleich) und BNE (Branch on Not Equal = Verzweige falls ungleich) bekommen hier also eine weitere Bedeutung. Selbstverständich können wir auch mit einer Konstanten vergleichen:
IF A = 3 THEN .... wird zu

```
LDA $1000  ; A                    LDY $1000  ;A
CMP #$03                          CPY #$03
BEQ ...    ;Verzweige, falls gleich  BEQ ....
```

Die verschiedenen Register unterscheiden sich hier nicht.

Von unseren BASIC-Beispielen ist noch eines ungelöst: IF D < 100 THEN ... Auch hier ist eine Lösung relativ einfach, wenn wir daran denken, daß der Prozessor intern eine Subtraktion durchführt und hinterher das Carry-Bit (komplementierte Borge-Flagge) entsprechend setzt. War der Registerinhalt kleiner als der Vergleichswert, so ist ein Borgen notwendig und das Carry-Bit wird gelöscht. Ist der Registerinhalt dagegen größer oder gleich dem Vergleichswert, so ist kein Borgen erforderlich und das Carry-Bit wird gesetzt. Mit dem Carry-Bit können wir damit zwei weitere Bedingungen testen: kleiner oder größer/gleich.

```
LDA $1003  ; D                    LDA $1003  ; D
CMP #100                          CMP #100
BCC ...    ; kleiner              BCS ....   ; gleich/größer
```

Wie Sie leicht sehen können, fehlen uns die Vergleiche „kleiner/gleich“ und „größer“. Sie lassen sich nur durch ein kombiniertes Abfragen der Flaggen lösen:

```
LDA $1003 ; D
CMP #$55
BCC ...   ; kleiner
BEQ ...   ; gleich
BCS ...   ; größer
```

Der Flaggentest muß in dieser Reihenfolge stattfinden, um eindeutig zu sein. Da es für den Anfänger sehr schwierig ist, sich zu merken, wann das Carry-Bit gesetzt ist und wann nicht, akzeptieren viele Assembler (incl. ASSESSOR) für BCC auch BLT (Branch on Lower Than = Verzweige falls kleiner) und für BCS

auch BGE (Branch on Greater than or Equal = Verzweige falls kleiner oder gleich), die sich leichter einprägen lassen.
Wenn Sie ein Ergebnis mit dem Carry-Bit testen wollen, sollten Sie nie mit $00 vergleichen, da jeder Wert (von $00 bis $FF) größer oder gleich $00 ist. Das Carry-Bit ist immer gesetzt.
Die interne Subtraktion beeinflußt noch eine weitere Flagge. Wenn der Registerinhalt kleiner als der Vergleichswert ist, liegt das Ergebnis unter Null, das Byte „klappt um". Auch wenn wir zur Zeit gar nicht mit vorzeichenbehafteten Zahlen rechnen, der Prozessor setzt eine weitere Flagge im Statusregister, die N- oder Negativ-Flagge, wenn das Bit 7 gesetzt wurde. Der Test der N-Flagge (der bei einigen amerikanischen Autoren sehr gebräuchlich ist) ist aber für unsere normalen Anwendungen nicht sehr aussagekräftig: Wenn der Akkumulator den Wert $10 enthält und gegen $11 verglichen wird, wird Bit 7 gesetzt und damit auch die N-Flagge. Ist der Vergleichswert aber sehr viel größer als der Akku (z.B. $FD), ist Bit 7 **nicht** gesetzt und damit die N-Flagge auch nicht, obwohl auch hier das Register kleiner als der Vergleichswert ist. Vergleichen wir $FA mit $20, so wird die N-Flagge gesetzt, obschon jetzt das Register größer als der Vergleichswert ist. **Die Logik der N-Flagge** (in der Tabelle eingeklammert) **gilt nur, wenn wir vorzeichenbehaftete Ganzzahlen** (signed Integers) **testen**.
Die Stellung der einzelnen Flaggen hat damit folgende Bedeutung:

	Register < Wert	Register = Wert	Register > Wert
N-Flagge:	(1)	0	(0)
Z-Flagge:	0	1	0
C-Flagge:	0	1	1

Noch eine kleine Anmerkung: Eine Verzweigung (Branch) führt nur dann zu einer Entscheidung, wenn das Sprungziel **nicht** der nächste Befehl ist. Folgender Code ist also sinnlos:

```
        LDA $1000
        CMP #$1F
        BEQ GLEICH   ;sinnlos!
GLEICH  LDA #$FF
        ....
```

In dem (falschen) Beispiel gelangen Sie immer zu „GLEICH", egal ob die Null-Flagge gesetzt ist oder nicht. Eine echte Entscheidung findet nur statt, wenn zwischen dem Verzweigungsbefehl (hier BEQ) und dem Sprungziel (hier GLEICH) Befehle stehen, die für den Fall gelten sollen, daß die Verzweigungsbedingung *nicht* zutrifft.

Lektion 17: Schleife vor und zurück

In Lektion 13 hatten wir schon einige Schleifen konstruiert. Diese liefen aber immer so lange, bis der Zähler Null geworden war. Zusammen mit den Vergleichsbefehlen haben wir jetzt neue Möglichkeiten, kleine und große Schleifen zu entwerfen. In BASIC kommt sehr häufig folgende Zeile vor:

```
XX  FOR I = 1 TO N: .... : NEXT I
```

Hier wird ein Schleifenzähler bei jedem Durchgang um 1 erhöht, bis der Endwert erreicht ist und die Schleife abgebrochen wird. In dem „FOR"-Teil wird dieser Zähler jeweils erhöht, im „NEXT"-Teil dann überprüft. In Assembler müssen wir diese Schritte einzeln programmieren, d.h. wir müssen zunächst einem Speicherplatz oder einem Register den Anfangswert des Zählers zuweisen, dann den Arbeitsteil der Schleife ausführen, den Zähler erhöhen, ihn überprüfen und schließlich entscheiden, ob wir wieder an den Schleifenbeginn zurückspringen. In BASIC sähe dieses etwa folgendermaßen aus:

```
100  I = 1
110  .....
120  I = I + 1
130  IF I < = N THEN 110
```

Für den Assembler-Programmierer ist die Abfrage in Zeile 130 ungünstig, da wir keinen einfachen Test auf „kleiner/gleich" zur Verfügung haben. Eine Möglichkeit wäre es, auf „kleiner N+1" zu testen. Wenn N eine Konstante ist, kann dies auch leicht bewerkstelligt werden. Folgende Schleife wird z.B. 5 mal durchlaufen:

```
          LDY #$01      ; Zähler gleich 1
SCHLEIFE  ....          ; Arbeitsteil
          INY           ; Zähler = Zähler + 1
          CPY #$06      ; < 6 ?  (N + 1)
          BLT SCHLEIFE  ;(= BCC) Ja
          ....          ; Nein
```

Diese Lösung ist nicht möglich, wenn N variabel ist und in irgendeiner Speicherstelle von einem anderen Programmteil definiert wird. Aber auch hier gibt es einen Ausweg: wir beginnen mit einem um 1 erniedrigten Schleifenzähler und

stellen den Inkrement-Befehl an den Anfang der Schleife. Dadurch wird gleich im ersten Durchgang der Zähler auf den richtigen Anfangswert gebracht. Da der Test erst am Ende der Schleife steht, wird der Arbeitsteil auch dann noch ausgeführt, wenn der Zähler bereits auf N steht. In BASIC sieht das dann so aus:

```
100 I = 0
110 I = I + 1
120 .....
130 IF I < N THEN 110
```

In Assembler geht es genauso (N sei in $1010 abgelegt):

```
          LDX #$00      ; Schleifenstart - 1
SCHLEIFE  INX           ; Zähler = Zähler + 1
          ....
          CPX $1010     ; < N ?
          BCC SCHLEIFE  ; Ja
          ....          ; Nein
```

Versuchen Sie einmal, das folgende kleine Programm zu assemblieren. Die Adresse $0500 liegt übrigens im (40 Z/Z) Textbildschirm, so daß Sie das Funktionieren gut überprüfen können.

```
          LDY #$41
          STY $500
SCHLEIF1  STY $501
          DEY
          BNE SCHLEIF1
SCHLEIF2  STY $502
          DEY
          CPY#$C0
          BNE SCHLEIF2
          RTS
```

Bevor Sie dieses Programm zum ersten Mal ausführen, versuchen Sie vorherzusagen, was es macht. Eine weitere Aufgabe besteht darin, folgendes BASIC-Programm zu übertragen:

```
100  FOR I = 1 TO N STEP 2
110  ....
120  NEXT I
```

Mit den bisher gelernten Befehlen sollte es nicht allzu schwer zu realisieren sein. Schleifen können aber auch etwas komplizierter ausfallen. Stellen Sie sich folgendes BASIC-Programm vor, in dem „J" bereits an anderer Stelle definiert wird:

```
100 I = 0
110 .....
120 I = I + 1
130 IF I + J < 10 THEN GOTO 110
```

In Assembler können wir dies so programmieren, wenn I = $1010 und J = $1011 ist:

```
          LDA #$00          ; I auf den Wert
          STA $1010         ; $00 setzen
SCHLEIFE  .....             ; Arbeitsteil
          INC $1010         ; I = I + 1
          LDA $1010         ; I laden
          CLC               ; Addition vorbereiten
          ADC $1011         ; + J
          CMP #10           ; < 10 ?
          BLT SCHLEIFE      ; Ja
```

Bisher haben wir unseren Schleifenzähler immer erhöht (inkrementiert) und dann verglichen. Es ist aber viel einfacher zu testen, ob eine Variable Null ist, als zu testen, ob sie kleiner oder gleich oder größer als ein Nichtnull-Wert ist. Viele Schleifen in Assembler werden deshalb „rückwärts" programmiert, wobei der Schleifenzähler bis auf Null herunter gezählt wird. Daß der Index dann mit N anfängt, ist für die meisten Anwendungen ohne Belang. Die einzelnen Schritte lauten dann:

Lade ein Register (X oder Y) mit der Anzahl der Durchgänge
Führe den Arbeitsteil der Schleife aus
Dekrementiere das Register (DEX oder DEY)
Wenn das Ergebnis <> Null ist, gehe zum 2. Schritt zurück (BNE).

```
          LDY #$40          ; Zahl der Durchgänge
SCHLEIFE  .....             ; Arbeitsteil
          DEY               ; Zähler - 1
          BNE SCHLEIFE      ; wenn <> Null weiter in der Schleife
```

Hätten Sie diese Schleife „vorwärts" aufgebaut, wäre nach jedem INY ein CPY erforderlich gewesen. Das kostet nicht nur Speicherplatz sondern auch Zeit. Assembler-Programme können bei gleicher Leistung also unterschiedlich schnell sein. In diesem Fall ist die schnellere Version auch noch die kürzere, ein Grund mehr, sie zu verwenden.

Lektion 18: Hast du da noch Töne

Unsere bisherigen Programme hatten fast alle den „Fehler", daß wir sie nicht mehr beeinflussen konnten, wenn sie erst einmal liefen. In aller Regel wollen wir aber Programme schreiben, die mit dem Benutzer in einen Dialog treten. Dazu gibt es zwei Grundvoraussetzungen: Der Rechner muß Ergebnisse, Fragen usw. in einer für den Benutzer erkennbaren Weise ausgeben und umgekehrt auf Eingaben reagieren.
Die übliche Eingabeeinheit ist die Tastatur. Wenn wir irgendeine Taste drükken, produziert sie ein bestimmtes Bitmuster und sendet dieses zum Rechner. Lediglich die Control-Taste (CTRL), die Umschalttaste (SHIFT) und die Feststelltaste (Caps lock) sowie bei älteren Apple die REPT-Taste erzeugen selber kein Signal, sondern beeinflussen das Ergebnis anderer Tasten. Jede Taste (außer den gerade genannten) sendet genau ein Byte, das entweder für einen Buchstaben (A–Z und a–z), eine Zahl (0–9) oder ein Sonderzeichen (z. B. *:+; usw. aber auch ÜÄÖüäöß) stehen kann. Schließlich gehören auch noch die Leertaste (Space) und ESCAPE dazu. Sie haben genauso ihren Code wie alle anderen, auch wenn wir sie normalerweise nicht „sehen". Die letzte Zeichenart bilden die Control-Zeichen, die dadurch entstehen, daß die CTRL-Taste gleichzeitig mit einer anderen Taste gedrückt wird. <CTRL-C> bedeutet, daß „CTRL" *und* „C" gedrückt werden, wodurch nur ein Byte erzeugt wird. Ein paar Control-Zeichen besitzen eine eigene Taste. So produziert der Rückwärtspfeil (Links) ein <CTRL-H>, der Vorwärtspfeil (Rechts) ein <CTRL-U>, die RETURN-Taste ein <CTRL-M>, der Aufwärtspfeil ein <CTRL-K>, der Abwärtspfeil ein <CTRL-J> und die TAB-Taste ein <CTRL-I>, ohne daß Sie die Control-Taste betätigen müßten.
Welcher Code von welcher Taste erzeugt wird, ist genormt. Es gibt weltweit zwei gebräuchliche Normen: ASCII und EBCDIC. EBCDIC steht für „Expanded Binary-Coded Decimal Interchange Code" und wird vornehmlich auf Großcomputern benutzt. Praktisch alle Home- und Personal-Computer (incl. Apple) benutzen den ASCII-Code (American Standard Code for Information Interchange = Amerikanischer Standard-Code für Informationsaustausch). Im Anhang finden Sie eine Tabelle mit dem vollständigen ASCII-Code. Der Apple hat allerdings noch einige Besonderheiten, da er gleichzeitig die Darstellungsform auf dem Bildschirm (NORMAL, INVERSE oder FLASH) mit codiert und so zu einer etwas eigentümlichen Einteilung des ASCII-Codes kommt. Auch hierzu finden Sie eine Tabelle im Anhang. Wir werden uns später noch näher damit beschäftigen.

Wie erfährt nun der Rechner (oder genauer der Prozessor), daß eine bestimmte Taste gedrückt wurde? Die Tastatur ist mit der Speicherstelle $C000 verbunden und jedes Byte kommt hier an. Wenn Sie sich die ASCII-Tabelle genauer ansehen, werden Sie feststellen, daß alle Zeichen zweimal darin vorkommen. Der ASCII-Code ist nämlich ein 7-Bit Code, bei dem nur die Bits 0 bis 6 das Zeichen bestimmen. Das 8. Bit (Bit 7) kann gesetzt sein oder auch nicht, es ändert nicht das Zeichen. Einige Rechner und Drucker benutzen Bit 7 für interne Unterscheidungen. Für den EPSON-Drucker bedeutet ein nicht gelöschtes Bit 7 beispielsweise gerade Schrift und ein gesetztes Bit 7 kursive Schrift.
In der Speicherstelle $C000 hat es mit dem Bit 7 eine andere Bewandnis: sobald eine Taste gedrückt wird, geht es auf 1. Es kann damit als Flagge dafür dienen, daß ein Tastendruck erfolgte. Auch wenn wir die Taste wieder loslassen, bleibt das Byte mit dem gesetzen Bit 7 in $C000 erhalten und wartet darauf, gelesen zu werden. Der folgende Tastendruck würde es allerdings überschreiben (außer bei externen Tastaturen mit Handshake). Nun muß es noch eine Möglichkeit geben, das Bit 7 wieder auf 0 zurückzusetzen, nachdem das Zeichen gelesen wurde. Wenn wir (einen beliebigen Wert) in die Speicherstelle $C010 schreiben (z.B. STA $C010) oder von dort lesen (z.B. LDA $C010), wird dadurch ein Signal (Strobe) ausgelöst, das Bit 7 von $C000 löscht.
Das folgende kleine Programm liest jeden Tastendruck und gibt das empfangene Zeichen auf dem Bildschirm aus. Dies geschieht der Einfachheit halber an einer festen Position.

```
TASTE  LDA $C000      ; Tastatureingabe lesen
       BPL TASTE      ; Schleife, bis Bit 7 gesetzt ist
       STA $C010      ; „Strobe„ von Bit 7
       STA $0528      ; ausgeben auf den Bildschirm
```

Besonders die Verzweigung ist hier interessant. Der erste Befehl (LDA) liest die Tastatureingabe. Wurde noch keine Taste gedrückt, so ist Bit 7 nicht gesetzt. Aus Lektion 5 wissen wir ja schon, daß Bytes mit gelöschtem Bit 7 als positiv gelten. Deshalb löscht der Prozessor im Statusregister nach dem LDA die Negativ-Flagge. Die bedingte Verzweigung BPL (Branch on PLus = Verzweige, wenn positiv) ist damit erfüllt und wird deshalb ausgeführt. Die beiden Befehle (LDA und BPL) bilden eine Schleife, die solange läuft, bis Bit 7 gesetzt ist und damit das Byte für den Apple als negativ gilt. Das bedeutet, daß bis zu einem Tastendruck gewartet wird. Wird die Taste gedrückt, geht Bit 7 und damit auch die N-Flagge auf 1, das Programm „fällt durch" die Verzweigung und führt den nächsten Befehl aus. Der Code der betätigten Taste bleibt dabei im Akkumulator. Er verändert sich auch nicht, wenn wir den Akku nach $C010

speichern und dadurch das Strobe-Signal auslösen, um anzuzeigen, daß wir die Information gelesen haben. Auf diese Weise wird das nächste Warten bis zu einem Tastendruck vorbereitet.
Nicht nur die Tastatur ist fest mit einer Speicherstelle verbunden. Zum Lautsprecher gehört die Adresse $C030. Wenn wir aus dieser Adresse einen (zufälligen) Wert lesen oder etwas dorthin schreiben, ergibt das einen „Tack" im Lautsprecher. Wiederholen wir diesen Vorgang sehr schnell hintereinander, vernehmen wir einen konstanten Ton, dessen Höhe von der Häufigkeit abhängt, mit der wir aus $C030 lesen. Dies darf wiederum aber auch nicht zu schnell geschehen, da dann die Membran des Lautsprechers nicht mehr „mitkommt".
Wir wollen nun ein kleines Programm schreiben, das bei jedem Tastendruck einen unterschiedlichen Ton von sich gibt. Wir werden dazu eine ganze Reihe von Befehlen kombinieren, die wir bisher gelernt haben.
Zunächst einmal das Programm, die Erläuterungen folgen danach:

```
                        1    ;************************
                        2    ;     Tasten-Musik      *
                        3    ;************************
                        4    ;
                        5              ORG  $300
                        6    ;
                        7    SPKR      EQU  $C030      ;Lautsprecher
                        8    STROBE    EQU  $C010      ;Strobe
                        9    KBD       EQU  $C000      ;Tastatur
                       10    ;
0300: AD 00 C0         11    START     LDA  KBD        ;Tastatur
0303: 10 FB            12              BPL  START
0305: 8D 10 C0         13              STA  STROBE
0308: 85 06            14              STA  $06
030A: 38               15              SEC
030B: E9 7F            16              SBC  #$7F
030D: 85 07            17              STA  $07
030F: A5 07            18    SCHLEIFE  LDA  $07
0311: 8D 30 C0         19              STA  SPKR
0314: C6 06            20              DEC  $06
0316: F0 E8            21              BEQ  START
0318: 38               22              SEC
0319: E9 01            23    MINUS     SBC  #$01       ;Verzögerung
031B: D0 FC            24              BNE  MINUS
031D: F0 F0            25              BEQ  SCHLEIFE

*** ENDE ***
```

Hier sehen wir schon das vollständig assemblierte Programm. Zwei Neuerungen sind noch hinzugekommen. Wir haben einige Zeilen eingefügt, die mit einem „;" beginnen. Dieses sind Kommentarzeilen (z.B. für den Titel), die nur zu unserer Information dienen, vom Assembler aber übergangen werden und

keinen Code erzeugen. In BASIC kennen Sie sicherlich die REM-Zeilen. Genau wie bei REM können Sie einen Kommentar auch an den Schluß einer Zeile hängen, wenn ihm ein „;“ voransteht. Alle Zeichen hinter „;“ werden ignoriert, dürfen also keinen Assembler-Code enthalten (bei den REM's ist es ja auch so).

Die zweite Neuerung ist der Befehl EQU. Sie werden ihn im Befehlssatz des 6502 (Kapitel 4) vergeblich suchen. EQU steht für „EQUates“, was soviel wie „Ist gleich“ bedeutet. Wir haben hier neben ORG einen weiteren Befehl kennengelernt, der den Assembler steuert und der nicht für den Prozessor gedacht ist.

Was macht nun EQU? Daß wir die Ziele von Verzweigungen innerhalb des Programms mit sogenannten Marken (Labels) angeben können, haben wir schon gelernt. Der Assembler berechnet dann selbsttätig die richtige Adresse oder den richtigen Offset. Wenn aber ein Ziel außerhalb des eigentlichen Programms liegt, dann müssen wir dem Assembler die dazugehörige Adresse mitteilen. Dies geschieht im allgemeinen in einem sog. Definitionsteil am Beginn des (Assembler-) Programms. Durch EQU wird einem Label (Marke) eine Adresse zugeteilt, die der Assembler dann im Programm einsetzen kann. Wenn Sie links auf den Maschinencode sehen, werden Sie erkennen, daß alle Label dort mit ihren Adressen aufgelöst sind. Über den Definitionsteil ist es möglich, Speicherstellen oder Unterroutinen einen sinnvollen Namen zu geben, unter dem wir sie dann im Programm aufrufen können. Viele Programmteile im ROM des Apple haben schon feste Namen, die wir nach Möglichkeit auch verwenden sollen, damit ein anderer Programmierer schnell erkennt, welche Routinen benutzt werden. Im zweiten Band werden wir uns ganz ausgiebig mit den ROM-Routinen befassen.

Das eigentliche Programm benutzt nur die uns bisher bekannten Befehle. Zunächst wird in einer Schleife die Tastatur solange gelesen, bis ein Tastendruck erfolgt ist. Ist das geschehen, wird über Strobe das Bit 7 von $C000 zurückgesetzt und der ASCII-Wert der Taste nach $0006 zwischengespeichert. Da Bit 7 immer gesetzt ist, liegen die möglichen Werte zwischen $80 (= <CTRL-@>) und $FF (<DEL>). Ziehen wir davon $7F ab (vorher Carry setzen, da kein alter Übertrag zu berücksichtigen ist!), liegen die Werte zwischen $01 und $80. Diese werden nach $0007 gespeichert und dienen als Verzögerungswerte, indem sie in den Akkumulator geladen (Zeile 18) und dann in Zeile 22/24 auf Null heruntergezählt werden, bevor in der größeren Schleife wieder der Lautsprecher betätigt wird. Gleichzeitig wird bei jedem Durchgang durch die große Schleife die Speicherstelle $0006 um 1 erniedrigt, bis auch hier der Wert 0 entsteht. Dann hört der Ton auf und das Programm verzweigt wieder zum Lesen

der Tastatur. Beenden können Sie das Programm nur, indem Sie <RESET> (oder <CTRL-RESET>) drücken.
Für Besitzer eines 65C02-Prozessors ist es möglich, die Zeilen 22 und 23 durch ein „DEA“ zu ersetzen. Dadurch ändert sich der Ton etwas, da SEC/SBC zusammen 4 Takte benötigen, DEA aber nur 2, so daß die Schleife jetzt schneller läuft.

Lektion 19: Wir setzen Zeichen

Wir haben bisher gesehen, daß ein Byte eine Ganzzahl, eine vorzeichenbehaftete Ganzzahl oder einen Teil einer 16-Bit (2-Byte) Zahl darstellen konnte. Eine weitere Interpretation ist die eines Bildschirmzeichens. Diese umfassen
a) Buchstaben (a...z, A...Z)
b) Zahlen (0...9)
c) Leerzeichen
d) Sonderzeichen (.,;:?#"!$ÄÖÜäöüß....)
Im Anhang finden Sie eine Tabelle, in der der gesamte Zeichenumfang des Apple mit den entsprechenden Byte-Werten dargestellt ist.
Die Zeichen können auf dem Bildschirm unterschiedlich dargestellt werden:
a) Normal (helle Zeichen auf dunklem Hintergrund)
b) Invers (dunkle Zeichen auf hellem Hintergrund)
c) Blinkend (abwechselnd Normal und Inverse, auch Flash genannt)
Inverse Kleinbuchstaben stehen nur auf dem IIe und IIc zur Verfügung, wenn auf blinkende Zeichen verzichtet wird. Der IIplus kennt ohne Umrüstsatz überhaupt keine Kleinbuchstaben. Beim IIc und beim „enhanced" IIe gibt es schließlich im alternativen Zeichensatz noch die sogenannten Mauszeichen, die kleine Grafiken bilden und nichts mit den üblichen ASCII-Zeichen zu tun haben.
Auf der Tastatur können sie zusätzlich noch Control-Zeichen erzeugen, die aber sämtlich nicht auf dem Bildschirm dargestellt werden. Einige haben jedoch eine besondere Bedeutung als Steuerzeichen, so daß ihre Wirkung zu sehen oder zu hören ist. So ist z.B. <CTRL-H> mit dem Rückwärtspfeil und <CTRL-M> mit der RETURN-Taste identisch.
Bitte beachten Sie, daß der Code für eine Ziffer nicht identisch mit der Ziffer ist. So gehört zur Ziffer „1" nicht der Code „1", sondern der Code „$B1". Die Codes für die zehn Ziffern liegen allerdings ihrerseits wieder hintereinander: „2" = „$B2", „3" = „$B3" usw.
Wir wollen jetzt ein Programm schreiben, das alle Zeichen des Apple auf dem Bildschirm ausgibt. Dazu wäre es schön, wenn zunächst der Bildschirm gelöscht würde. In BASIC befehlen Sie ganz einfach „HOME". Wenn wir in Maschinensprache den Schirm löschen wollen, müssen wir in alle Plätze des Bildschirmspeichers Leerzeichen schreiben. Wie Sie aus dem Anhang entnehmen können, haben Leerzeichen den Code $A0. Der Bildschirmspeicher (40 Z/Z) reicht von $0400 bis $07FF im Hauptspeicher des Apple. Das sind 4 mal 256 ($FF)

Zeichen. Wir könnten eine Schleife schreiben, die an alle Positionen zwischen $0400 und $07FF ein $A0 schreiben würde. Viel einfacher ist es aber, die eingebaute „HOME"-Funktion des Apple zu benutzen. Wir können sie allerdings nicht wie in Basic nur mit „HOME" aufrufen, sondern müssen sie als Unterprogramm an ihrer Speicheradresse aufrufen. Wir finden die „HOME"-Funktion ab $FC58 im Monitor-ROM. In BASIC rufen wir ein Unterprogramm mit „GOSUB ..." auf und kehren mit „RETURN" wieder zum Hauptprogramm zurück. In Assembler ist es ganz genauso. Statt „GOSUB" heißt es dort „Jump to SubRoutine" oder als Mnemonic „JSR". Das Unterprogramm endet mit dem Befehl „RTS". Die HOME-Routine unterscheidet sich je nach Monitor etwas. Wichtig ist aber, daß sie immer bei $FC58 beginnt und mit „RTS" endet. Ihr genaues Funktionieren liegt zur Zeit noch jenseits von unseren Kenntnissen und Möglichkeiten.

Die zweite neue Aufgabe ist es, Zeichen auf dem Bildschirm fortlaufend auszugeben. Auch hier hat der Apple für uns schon eine Routine parat. Sie heißt „COUT" und beginnt bei $FDED. COUT ist ein Kurzwort für „Charakter OUTput" (Zeichen-Ausgabe). Auch COUT können wir mit einem JSR aufrufen. Verglichen mit HOME tritt hier allerdings noch eine Neuerung auf: wir müssen aus dem Hauptprogramm eine Information an das Unterprogramm COUT übergeben. COUT erwartet, daß das auszugebende Zeichen beim Aufruf im Akkumulator steht.

Jetzt können wir uns daran machen, unser Programm zu schreiben. Wir müssen dem Assembler dabei die Startadressen von HOME und COUT mitteilen, da sie ja nicht im Programm selber liegen. Wir benutzen hier wieder den Pseudo-Opcode „EQU". Wenn wir erst definieren: COUT EQU $FDED, dann meint JSR COUT dasselbe wie JSR $FDED.

```
                    1   ;*******************************
                    2   ;  Zeichensatz - Demonstration  *
                    3   ;*******************************
                    4   ;
                    5   ;  Definitionen
                    6   ;
                    7   HOME      EQU $FC58
                    8   COUT      EQU $FDED
                    9   ;
                    10            ORG $300
                    11  ;
                    12  ;  Programm
                    13  ;
0300: 20 58 FC      14            JSR HOME
0303: A9 00         15            LDA #$00
```

```
0305: 20 ED FD   16   SCHLEIFE  JSR COUT
0308: 18         17             CLC
0309: 69 01      18             ADC #$01
030B: 90 F8      19             BCC SCHLEIFE
030D: 60         20             RTS

** ENDE **
```

Unser Programm wurde wieder für alle Apple geschrieben. Mit dem 65C02 können Sie die Zeilen 17 bis 20 ersetzen durch

```
 ....
0308:  1A        17             INA
0309:  D0 FA     18             BNE SCHLEIFE
030B:  60        19             RTS
```

Nach INA können wir keinen Übertrag mit dem Carry-Bit testen, da Inkrement und Dekrement diese Flagge nicht setzen. Stattdessen testen wir die Null-Flagge, die uns in diesem Fall auch einen vollständigen Durchgang anzeigt.
Wenn Sie einen Apple IIe oder IIc besitzen, versuchen Sie abwechselnd ein „POKE 49166,0" und ein „POKE 49167,0", bevor Sie die Zeichensatz-Demonstration starten. Wenn sich das Maschinenprogramm schon im Speicher befindet, gelingt das mit einem „CALL 768". Sie sollten dann entweder den normalen oder den alternativen Zeichensatz sehen. Bei einem Apple IIplus ist kein Unterschied sichtbar.
Zum Abschluß dieser Lektion schauen wir uns das Listing des Assemblers noch intensiver an. Ganz links steht immer eine vierstellige (hexadezimale) Adresse. Nach dem Doppelpunkt folgen zwischen 1 und 3 Bytes, die den erzeugten Maschinencode darstellen. Dahinter erkennen Sie den von Ihnen eingegebenen Quellcode. Manche Befehle erzeugen nur ein Byte an Code (z.B. RTS oder INA), in dem alle Information enthalten ist. Die Verzweigungen brauchen zwei Bytes: eines für den Branch-Befehl und eines für das Distanzbyte. Die beiden JSRs schließlich benötigen sogar 3 Bytes: eines für den Befehl selber und zwei für die Adresse der Unterroutine. Wenn Sie sich den Code genau ansehen, werden Sie feststellen, daß der höher- und der niederwertige Anteil der Adresse genau vertauscht im Maschinencode erscheinen. Das ist eine Eigenart des Prozessors, der erst immer das Lo-Byte und dann das Hi-Byte lesen muß. Um diese Vertauschung brauchen Sie sich aber nicht zu kümmern, da sie vom Assembler für Sie erledigt wird. Der Assembler zählt auch die Speicherplätze automatisch weiter, so daß die Adresse links immer zum ersten Byte der Zeile gehört. Für ein eventuell folgendes 2. und 3. Byte müssen Sie dann eine 1 oder 2 addieren, um dessen Lage im Programmspeicher zu erhalten.

Lektion 20: Ein guter Abgang

Zu den häufigsten Techniken gehört es, irgendeinen Text auf dem Bildschirm zu drucken. Mit unseren bisherigen Befehlen sind wir dazu bereits in der Lage. Wir schreiben jetzt das Äquivalent zu den BASIC-Zeilen
10 HOME
20 PRINT "MEIN ERSTER SATZ"
In der Maschinensprache gibt es viele Möglichkeiten, diesen Satz auszudrucken. Wir befassen uns jetzt mit einer recht einfachen Lösung. Im zweiten Band folgen dann noch ein paar raffiniertere Methoden.
Unsere Aufgabe besteht also darin, eine Zeichenkette auszugeben. Wie wir ein einzelnes Zeichen mit COUT auf dem Bildschirm bringen können, haben wir bereits gesehen. Wenn wir eine Zeichenkette drucken wollen, müssen wir nacheinander alle Zeichen in den Akkumulator laden und dann jeweils COUT aufrufen. Das ist ein typischer Anwendungsfall für eine Schleife. Unsere Zeichenkette müssen wir in ihrer ASCII-Form irgendwo im Speicher ablegen. Der Assembler hat dazu einen speziellen Befehl: ASC, das für ASCII-Daten steht. Als OPERANDen schreiben wir, wie in BASIC eingeschlossen von Anführungszeichen, unsere Zeichenkette. Nun müssen wir nacheinander die Zeichen der Kette in den Akkumulator bringen und ausgeben. Jedesmal rücken wir eine Position weiter.
Das läßt sich besonders schön mit der **indizierten Adressierung** verwirklichen. Sie kennen diese Art der Adressenbeschreibung aus dem Alltag auch: „Der Bäcker ist im fünften Haus hinter der Apotheke". Die Apotheke ist eine uns bekannte Basisadresse, zu der wir dann einen bestimmten Indexwert (hier 5) addieren, um zur endgültigen oder effektiven Adresse des Bäckers zu gelangen. In der Assemblersprache steht der Index entweder im X- oder im Y-Register, die genau aus diesem Grunde auch Indexregister heißen. Nehmen wir einmal an, X würde $05 enthalten. Dann würde der Befehl
LDA $0800,X
den Akkumulator mit dem Inhalt der Speicherstelle $0800 + $05 = $0805 laden. Mit dem Y-Register funktioniert das genauso, nur daß wir hier ein ‚Y' an die Basisadresse hängen.
Wollen Sie das X- oder Y-Register selber laden, so ist natürlich nur das andere Register als Index zugelassen:

```
LDX $1000,Y
LDY $23F0,X
```

Die indizierte Adressierung existiert zum Speichern nur für den Akkumulator. Wenn Sie eine der komplizierteren Adressierungsarten anwenden wollen, vergewissern Sie sich am besten im Kapitel 4, ob dieser Befehl überhaupt existiert. Der Assembler weist eine nicht erlaubte Adressierungsart zurück.
Wenn uns die Basisadresse nicht bekannt ist, können wir auch ein LABEL dafür einsetzen. Wir haben jetzt nur noch ein Problem zu lösen: woran sollen wir erkennen, daß die Zeichenkette zu Ende ist? Die bequemste Lösung besteht darin, als letztes Zeichen einen normalerweise nicht vorkommenden Code zu nehmen und mit einem Vergleich zu testen, ob er schon erreicht ist. Es hat sich eingebürgert, die Hexzahl $00 dafür zu nehmen, da sie besonders einfach zu testen ist: wird $00 in den Akkumulator geladen, so wird dadurch die Null-Flagge gesetzt. Wie aber bringen wir $00 in den Speicher? Mit ASC geht es nicht, da $00 kein druckendes ASCII-Zeichen ist. Der Assembler hat für solche Fälle einen eigenen Pseudo-Opcode: HEX, das für HEXadezimale Daten steht.
Nach soviel Theorie jetzt unser Programm:

```
                    1   ;*************************
                    2   ; Zeichenketten - Ausgabe *
                    3   ;*************************
                    4   ;
                    5   ; Definitionen
                    6   ;
                    7   HOME      EQU $FC58
                    8   COUT      EQU $FDED
                    9   ;
                   10             ORG $300
                   11  ;
                   12  ; Programm
                   13  ;
0300: 20 58 FC     14             JSR HOME
0303: A0 00        15             LDY #$00        ;initialisieren
0305: B9 11 03     16  SCHLEIFE   LDA TEXT,Y
0308: F0 06        17             BEQ ENDE        ;$00 gefunden
030A: 20 ED FD     18             JSR COUT        ;ausgeben
030D: C8           19             INY             ;nächstes Zeichen
030E: D0 F5        20             BNE SCHLEIFE
0310: 60           21  ENDE       RTS
                   22  ;
0311: CD C5 C9     23  TEXT       ASC "MEIN ERSTER SATZ"
0321: 00           24             HEX 00

** ENDE **
```

Mit dem Simulator IDUS können Sie sich gut ansehen, wie die Schleife und der Schleifenabbruch funktionieren. Sie können das Programm aber auch einfach mit „BRUN“ starten. Zunächst wird der Bildschirm gelöscht und dann oben

links „MEIN ERSTER SATZ“ gedruckt. Nun geschieht allerdings etwas Unerwartetes: wir kommen nicht nach Applesoft zurück, sondern der Rechner „hängt“, so daß Sie <RESET> drücken müssen. Sie haben keinen Fehler gemacht und auch das Programm hat ansich keinen „BUG“. Der Übeltäter ist hier DOS 3.3. Der Befehl RTS am Programmende sollte eigentlich zum Aufrufer, hier also zu Applesoft zurückführen. Wenn in einem Programm bei aktivem DOS jedoch Texte über COUT ausgedruckt werden, dann bringt DOS die Rücksprungadressen durcheinander. Wenn Sie in den Monitor gehen (CALL-151) und dann mit 300G unser Programm starten, funktioniert alles wie geplant und Sie landen wieder im Monitor. Aber auch für Applesoft und DOS 3.3 gibt es eine Lösung. Statt mit RTS zu schließen, springen wir zur Warmstart-adresse von DOS. Hierzu gibt es auf der Seite $03 einen Zeiger bei $03D0, von dem aus dann zur eigentlichen Routine verzweigt wird. In BASIC führen wir Sprünge mit „GOTO ...“ aus, in Assembler heißt der Befehl „JuMP“ (Springe) oder als Mnemonic „JMP“. Ersetzen Sie in Zeile 21 das RTS durch JMP $03D0, und der gute Abgang ist gesichert.

Lektion 21: Links herum und rechts herum

Wenn Sie im Dezimalsystem an das Ende einer Zahl eine Null anhängen, entspricht das einer Multiplikation mit 10. Wenn wir im Dualsystem eine Null anhängen, läuft dies auf eine Multiplikation mit 2 heraus. Jede Multiplikation ganzer Zahlen läßt sich im Prinzip auch durch Additionen ausführen. Aus 2 * 23 = 46 läßt sich so 23 + 23 = 46 machen. Mit Binärzahlen sieht das folgendermaßen aus:

```
  %00010111   ($17, 23)
+ %00010111   ($17, 23)
------------  ---------
  %00101110   ($2E, 46)
```

Wenn wir nur ein Byte betrachten, erreichen wir die Multiplikation mit 2, indem wir alle Bits nach links schieben, so daß das alte Bit 7 herausfällt. In das freiwerdende Bit 0 wird eine 0 hineingeschoben. Der 6502-Prozessor besitzt für diese Operation den Befehl ASL (Arithmetic Shift Left , Arithmetische Verschiebung nach links). Das herausgefallene Bit 7 geht nicht endgültig verloren, sondern wird im Carrybit aufbewahrt und zeigt, wenn es gesetzt ist, an, daß durch die Multiplikation das Fassungsvermögen des Bytes überschritten wurde.

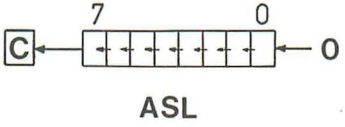


Abb. 13: Arithmetische Verschiebung nach links

Die Division durch 2 verläuft ganz ähnlich: hier müssen alle Bits um eine Stelle nach rechts wandern, und ganz links muß eine Null hineingeschoben werden. Der Befehl dazu lautet LSR (Logical Shift Right, Logische Verschiebung nach rechts). Diesmal landet das alte Bit 0 in der Übertragsflagge.

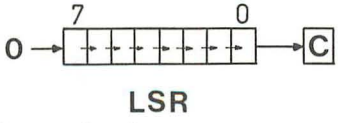


Abb. 14: Logische Verschiebung nach rechts

Wenn Sie eine ungerade Zahl mit LSR dividieren, erhalten sie immer ein abgerundetes Ergebnis:

%00010111 —> %00001011
$17, 23 $0B, 11

Jede ungerade Zahl hat Bit 0 gesetzt. Nach dem LSR befindet sich dieses Bit in der Übertragsflagge. Das ermöglicht einen einfachen Test auf gerade/ungerade:

```
LDA Zahl    ; die Zahl laden      LDA ZAHL
LSR         ; :2                  LSR
BCC GERADE  ;                     BCS UNGERADE
```

Wenn Sie ein aufgerundetes Ergebnis wünschen, läßt sich dieses leicht bewerkstelligen:

```
LDA Zahl    ; die Zahl laden
LSR         ; :2
ADC #$00    ; + 0 bei C = 0, +1 bei C = 1
```

War die Zahl ursprünglich gerade, so ist das Carrybit nach dem LSR nicht gesetzt und ADC #$00 verändert das Ergebnis nicht. War die Zahl aber ungerade, so wird bei ADC #$00 durch das dann gesetzte Carrybit eine 1 addiert und damit aufgerundet.
LSR und ASL beeinflussen aber nicht nur das Carrybit, auch die Null- und die Negativ-Flagge werden je nach Ergebnis der Operation eingerichtet.
Viele andere Multiplikationen und Divisionen lassen sich so bewerkstelligen. Durch dreimaliges Linksshiften erhält man eine Multiplikation mit 8. Ist der Faktor aber keine Zweierpotenz, wird es komplizierter. Weil die Multiplikation mit 10 relativ häufig vorkommt, wollen wir sie noch üben. Alle anderen Berechnungen führen wir mit allgemeinen Multiplikations- und Divisionsroutinen durch, die in den nächsten Lektionen folgen.
Versuchen wir es zunächst mit einem dezimalen Beispiel: $10 * 3 = 30$. Diese Rechnung können wir folgendermaßen zerlegen:
$10 * 3 = (2 * 3) + (8 * 3)$.
Wir erhalten so wieder Faktoren, die aus Zweierpotenzen bestehen. In einem Programm sieht das dann so aus:

```
LDA ZAHL    ; die Zahl laden
ASL         ; * 2
STA TEMP    ; 2 * ZAHL zwischenspeichern
ASL         ; * 2 (insgesamt * 4)
ASL         ; * 2 (insgesamt * 8)
ADC TEMP    ; 8 * ZAHL + 2 * Zahl = 10 * Zahl
```

TEMP ist irgendein von Ihnen gewählter Speicherplatz, um das Zwischenergebnis aufzubewahren. Vor dem Befehl ADC ist hier kein CLC nötig, da durch das voranstehende ASL das Carrybit noch gelöscht sein sollte. Ist das nicht der Fall, so ist schon die Multiplikation mit 8 ungültig und damit gleichzeitig auch die Addition. Eine Bereichsüberprüfung findet nicht statt und Sie als Programmierer sind dafür verantwortlich, daß die Ausgangszahl nicht den zulässigen Bereich überschreitet.

Lektion 22: Kreisverkehr

Unsere Rechenkünste in Lektion 21 waren auf nur ein Byte begrenzt. Was aber machen Sie, wenn Sie eine 16-Bit Zahl mit 2 multiplizieren wollen? Sie müssen das niederwertige und das höherwertige Byte getrennt nacheinander nach links shiften. Das Problem liegt darin, wie Sie den Übertrag von Lo-Byte, der sich im Carrybit befindet, auf einfache Weise in das Hi-Byte hineinbekommen.

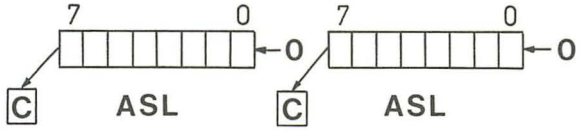


Abb. 15: 2-Byte ASL

Ideal wäre ein Befehl, der im Hi-Byte alle Bits nach links shiftet und in das Bit 0 nicht eine 0, sondern den Inhalt des Carrybits transportiert. Der 6502 besitzt einen solchen Befehl: ROL (ROtate Left, Rotiere nach links). Das alte Bit 7 des Hi-Bytes landet danach im Carry. ROL vollzieht eine vollständige Kreisbewegung, daher auch der Name „Rotiere".

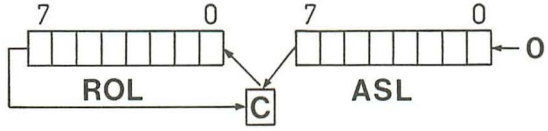


Abb. 16: Doppelshift

Eine 16-Bit-Zahl in den Speicherstellen Zahl-Lo und Zahl-Hi können wir dann mit

```
ASL Zahl-Lo
ROL Zahl-Hi
```

um eine Position nach links shiften und so mit 2 multiplizieren. Diese Operation wird auch Doppelshift genannt.
Für die umgekehrte Richtung steht uns der Befehl ROR (ROtate Right, Rotiere nach rechts) zur Verfügung.

```
LSR Zahl-Hi
ROR Zahl-Lo
```

Beachten Sie, daß Sie beim Doppelshift nach links mit dem Lo-Byte und beim Doppelshift nach rechts mit dem Hi-Byte anfangen müssen. Wenn wir um 2 Positionen shiften wollen, müssen wir abwechselnd das Hi- und das Lo-Byte bewegen:

```
ASL Zahl-Lo
ROL Zahl-Hi     ; 2facher Doppelshift nach links
ASL Zahl-Lo
ROL Zahl-Hi
```

Lektion 23: Keine wundersame Vermehrung

Unsere soeben gelernten Verschiebungs- und Rotationsbefehle werden wir jetzt verwenden, um eine richtige Multiplikationsroutine zu schreiben. Schauen wir uns zunächst noch einmal an, wie wir im Dezimalsystem mit Papier und Bleistift multiplizieren. Wir schreiben den Multiplikanden und den Multiplikator nebeneinander und führen jetzt für jede Stelle des Multiplikators eine Multiplikation aus. Jedes Teilprodukt wird unter die betreffende Stelle des Multiplikators geschrieben. Wenn alle Teilprodukte berechnet sind, addieren wir sie zum Gesamtergebnis auf. Der Deutlichkeit halber ist im folgenden Beispiel auch eine Reihe ausschließlich mit Nullen geschrieben worden, die wir in der Praxis normalerweise nicht aufschreiben.

```
Multiplikand   Multiplikator
      123 * 102
      ---------
          123
           000        Teilprodukte
            246
          -----
          12546       Endergebnis
```

Die Multiplikation im Dualsystem geschieht genauso. Sie ist sogar noch einfacher, da die Teilprodukte nur aus der Multiplikation mit 0 oder 1 herrühren. Das Teilprodukt besteht daher entweder aus lauter Nullen oder aus dem Multiplikanden.

```
Multiplikand   Multiplikator
    01111011 * 01100110
    -------------------
        00000000
         01111011
          01111011
           00000000
            00000000            Teilprodukte
             01111011
              01111011
               00000000
      -----------------
      0011000100000010          Endergebnis
```

Wie Sie sehen, ist das Ergebnis bedeutend länger als die beiden Ausgangswerte. Allgemein gilt, daß die Länge des Produkts höchstens so groß ist wie die Summe der Längen von Multiplikand und Multiplikator. Wenn wir zwei 8-Bit-Zahlen multiplizieren, müssen wir also für das Ergebnis 16 Bit bereitstellen.

Diesen Vorgang setzen wir jetzt in ein Programm um. Wenn wir den Multiplikator in einer Schleife insgesamt 8-mal nach links shiften, erscheinen alle seine Bitstellen nacheinander im Carrybit und wir können mit BCC testen, ob das Bit gesetzt war oder nicht. Wenn es 0 war, müssen wir nichts weiter tun (+ 00000000), wenn es 1 war, müssen wir den Multiplikanden zu unseren Zwischenergebnissen addieren. Dabei ist darauf zu achten, daß auch das Zwischenergebnis jedesmal um eine Stelle nach links verschoben wird, wenn der Multiplikator bewegt wurde. Da das Additionsergebnis größer als 1 Byte werden kann, müssen wir auf Übertrag (C = 1) testen und gegebenenfalls zum höherwertigen Byte des Ergebnisses eine 1 addieren. Die folgende Routine wendet noch einen kleinen Trick an: da der Multiplikator nach links geschoben wird, werden seine rechten Stellen langsam frei. In diese freien Positionen werden die Überträge in das Hi-Byte des Ergebnisses geschrieben. Nach acht „Runden" steht dann das höherwertige Byte des Ergebnisses im ehemaligen Multiplikator. Auf diese Weise wird eine Speicherstelle eingespart und gleichzeitig werden der Multiplikator und das höherwertige Ergebnis mit nur einem Befehl nach links bewegt. Damit Sie die Multiplikationsroutine universell als Unterprogramm verwenden können, vereinbaren wir auch noch eine Wertübergabe: beim Aufruf müssen Multiplikand und Multiplikator im Akkumulator bzw. Y-Register stehen, das Ergebnis wird mit dem Lo-Byte im Akkumulator und dem Hi-Byte im Y-Register wieder zurückgegeben. Für Ihre eigenen Programme können Sie natürlich einen anderen ORG als $0300 nehmen und die Routine noch einmal assemblieren.

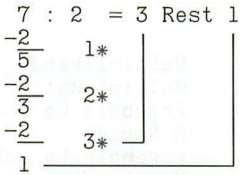
Das folgende Listing ist reich kommentiert, damit Sie es besser nachvollziehen können, denn eine so knappe Multiplikationsroutine gehört schon zur gehobenen Klasse von Assemblerprogrammen. Es ist auch sinnvoll, sich mit IDUS die einzelnen Schritte genau anzusehen. Sie können den Akkumulator und das Y-Register mit verschiedenen Werten laden und dann jeweils bei $300 starten. Eine Erklärung bin ich Ihnen noch schuldig: in den Zeilen 29 und 30 finden Sie einen neuen Pseudo-Opcode: DFS, das für „DeFine Storage" oder „Reserviere Speicherplatz" steht. Mit ihm läßt sich die im OPERANDEN angegebene Anzahl an Speicherplätzen im Code freihalten. Hier brauchen wir für den Multiplikanden und den Multiplikator je 1 Byte, die direkt hinter der eigentlichen Routine liegen.

```
                        1    ;************************
                        2    ; 8-Bit Multiplikation  *
                        3    ;************************
                        4    ;
                        5    ; Multiplikand im Akkumulator
                        6    ; Multiplikator im Y-Register
                        7    ; Produkt im Akku (Lo) und Y-Reg. (Hi)
                        8    ;
                        9            ORG $300
                       10    ;
                       11    ; Programm
                       12    ;
0300: 8D 20 03         13    MULT      STA MKAND       ;Multiplikand
0303: 8C 21 03         14              STY MATOR       ;Multiplikator
0306: A9 00            15              LDA #$00        ;Ergebnis Lo löschen
0308: A0 08            16              LDY #$08        ;8 Runden
030A: 0A               17    SCHLEIFE  ASL             ;Ergebnis Lo schieben
030B: 2E 21 03         18              ROL MATOR       ;M-kator/Ergebnis Hi
030E: 90 09            19              BCC WEITER      ;Bit testen
0310: 18               20              CLC             ;war 1, also
0311: 6D 20 03         21              ADC MKAND       ;+ Multiplikanden
0314: 90 03            22              BCC WEITER      ;mit Übertrag?
0316: EE 21 03         23              INC MATOR       ;Ja, +1 im Hi-Byte
0319: 88               24    WEITER    DEY             ;Schleifenzähler
031A: D0 EE            25              BNE SCHLEIFE    ;noch nicht fertig
031C: AC 21 03         26              LDY MATOR       ;Hi-Byte laden
031F: 60               27              RTS             ;zurück
                       28    ;
                       29    MKAND     DFS 1
                       30    MATOR     DFS 1           ;Zwischenspeicher

** ENDE **
```

Lektion 24: Teile und herrsche

Von den vier Grundrechenarten fehlt uns jetzt nur noch die Division. Divisionen lassen sich auf wiederholte Subtraktionen zurückführen:



Neben dem Quotienten kann bei der Division auch ein nichtteilbarer Rest auftreten. Wir können ihn entweder unberücksichtigt lassen oder ihn für die Modulo-Funktion einsetzen. MOD (= Modulo) ist nicht in Applesoft, aber im älteren Integer-BASIC enthalten.
Wir werden unsere Divisionsroutine mit einem 16-Bit Dividend beginnen, dessen Hi-Byte im Akkumulator und dessen Lo-Byte im Y-Register übergeben werden. Der 8-Bit Divisor kommt in das X-Register. Der Quotient wird im X-Register zurückgegeben, der Rest steht im Akkumulator. Wie schon bei der Multiplikationsroutine wird auch hier ein Byte für zwei Zwecke benutzt: in DEND (s.u.) befindet sich zunächst das niederwertige Byte des Dividenden. Wenn es nach links geshiftet wird, wird von rechts der Divisor hereingeschoben, so daß am Ende der Schleife nur noch der Divisor in DEND steht.
Unser Programm hat noch eine Besonderheit: die Routine kehrt nach dem RTS mit einem gesetzten Carrybit zurück, wenn ein Fehler aufgetreten ist. Dafür kann es zwei Gründe geben. Entweder haben Sie versucht, durch Null zu dividieren, oder aber der Quotient paßt nicht in ein Byte. Das Carrybit ist gelöscht, wenn alles in Ordnung ist. Ein Aufruf der Divisionsroutine sollte also folgendermaßen aussehen:

```
LDA DIVIDEND-HI
LDY DIVIDEND-LO
LDX DIVISOR
JSR DIVI
BCS FEHLER
...
```

Wenn Sie nur eine 8-Bit Zahl dividieren wollen, müssen Sie den Akkumulator mit dem Wert $00 laden!

```
                        1    ;**************************
                        2    ; Division 16 Bit / 8 Bit *
                        3    ;**************************
                        4    ;
                        5    ; Dividend in A (Hi) und Y (Lo)
                        6    ; Divisor in X
                        7    ; Quotient in X, Rest in A
                        8    ; Carry gesetzt bei Fehler
                        9    ;
                        10            ORG $300
                        11   ;
                        12   ; Programm
                        13   ;
0300: 8C 26 03          14   DIVI      STY DEND        ;Dividend Lo-Byte
0303: 8E 27 03          15            STX DOR         ;Divisor
0306: CD 27 03          16            CMP DOR         ;Zu groß oder durch 0?
0309: B0 1A             17            BCS DIVEND      ;Ja, Fehler
030B: A2 08             18            LDX #$08        ;8 Runden
030D: 0E 26 03          19   SCHLEIFE  ASL DEND        ;Doppelshift des
0310: 2A                20            ROL             ;Dividenden
0311: B0 05             21            BCS SUBTR       ;Subtrah. bei C=1
0313: CD 27 03          22            CMP DOR         ;mit Divisor vergl.
0316: 90 06             23            BCC WEITER      ;ist kleiner
0318: ED 27 03          24   SUBTR     SBC DOR         ;Divisor subtrahieren
031B: EE 26 03          25            INC DEND        ;+1 zum Quotienten
031E: CA                26   WEITER    DEX             ;Schleife zu Ende?
031F: D0 EC             27            BNE SCHLEIFE    ;Nein, weitermachen
0321: AE 26 03          28            LDX DEND
0324: 18                29            CLC             ;ohne Fehler
0325: 60                30   DIVEND    RTS
                        31   ;
                        32   DEND      DFS 1
                        33   DOR       DFS 1

** ENDE **
```

Auch diesmal finden Sie viele Kommentare, denn die Division ist noch eine Spur schwieriger als die Multiplikation. Sie sollten auf eine Simulation mit IDUS nicht verzichten!

Zunächst werden der Divisor und das Lo-Byte des Dividenden zwischengespeichert. Dann wird das Hi-Byte mit dem Divisor verglichen. Wenn es gleich groß oder größer ist, ist der Divisor entweder Null oder das Ergebnis würde zu groß werden. In beiden Fällen verzweigen wir (BCS) mit gesetztem Carrybit zum Ende der Routine.

Mit dem X-Register wird wie bei der Multiplikation eine Schleife aufgebaut, die zwischen Zeile 19 und Zeile 27 liegt. Da unser Dividend 16-Bit groß ist, brauchen wir zwei Speicherstellen, in denen die Subtraktionen stattfinden: den Akkumulator und DEND. Mit einem Doppelshift (ASL DEND, ROL) werden diese beiden bei jedem Schleifendurchgang um eine Position nach links versch-

ben. Wenn dabei das Carrybit gesetzt wird, wird der Divisor vom Akkumulator abgezogen. Ist das Carrybit gelöscht, wird mit CMP DOR zunächst überprüft, ob der Akkumulator größer oder gleich dem Divisor ist. Ist das der Fall, wird auch jetzt der Divisor vom Akkumulator subtrahiert. Für jede erfolgte Subtraktion wird mittels INC (Zeile 25) der Quotient um 1 erhöht. Der Quotient wird in den freiwerdenden Bits von DEND gebildet. Nach 8 Runden verbleibt im Akkumulator der Rest. Das X-Register wird mit dem Quotienten geladen und das Carrybit gelöscht, um die fehlerfreie Abarbeitung anzuzeigen.
Vor dem Befehl SBC in Zeile 24 muß nicht mittels SEC das Carrybit extra gesetzt werden, da es in jedem Fall bereits gesetzt ist: kommen wir von Zeile 21, so findet die Verzweigung (BCS) nur bei gesetztem Carry statt. Kommen wir von Zeile 23, so fällt das Programm nur dann durch den Befehl BCC, wenn das Carrybit gesetzt ist.

Lektion 25: Gesetzt oder nicht?

Die Shift-Befehle sind nicht nur für arithmetische Berechnungen wichtig, sie eignen sich auch vorzüglich, um einzelne Bits in einem Byte zu testen. Lassen Sie uns jetzt eine Routine schreiben, die ein Byte in Binärform auf dem Bildschirm ausgibt. Dazu müssen wir es in 8 Zeichen („0" oder „1") umwandeln, die nacheinander gedruckt werden. Die Ausgabe besorgt wieder die Monitor-Routine COUT für uns. Da diese nur von links nach rechts schreibt, müssen wir auch unser Byte von links nach rechts abarbeiten. Mit ASL shiften wir es nach links und testen nach jedem Schritt das Carrybit. Ist es gesetzt, war das Bit eine 1, ist es gelöscht, war das Bit eine 0. Auf Grund dieser Entscheidung laden wir entweder den ASCII-Wert für die Ziffer 0 (= $B0) oder für die Ziffer 1 (= $B1) in den Akkumulator und geben ihn über COUT aus. Mit dem Assembler können Sie einen ASCII-Wert direkt laden, wenn Sie nach dem Nummernkreuz (für die unmittelbare Adressierung) das Zeichen in Anführungszeichen setzen.

```
                        1   ;*************
                        2   ; Print Bits *
                        3   ;*************
                        4   ;
                        5   ; Byte im Akkumulator wird
                        6   ; als Binärzahl ausgegeben
                        7   ;
                        8   ;  Definitionen
                        9   ;
                        10  COUT      EQU $FDED
                        11  ;
                        12            ORG $300
                        13  ;
                        14  ; Programm
                        15  ;
0300: 8D 15 03          16  PRBITS    STA AKKU        ;Zwischenspeicher
0303: A2 08             17            LDX #$08        ; 8 Runden
0305: 0E 15 03          18  SCHLEIFE  ASL AKKU        ;nächstes Bit
0308: A9 B0             19            LDA #"0"        ;erst eine „0" laden
030A: 90 02             20            BCC OUTPUT      ;wirklich Null
030C: A9 B1             21            LDA #"1"        ;zu „1" verbessern
030E: 20 ED FD          22  OUTPUT    JSR COUT        ;ausgeben
0311: CA                23            DEX             ;Schleifenzähler
0312: D0 F1             24            BNE SCHLEIFE
0314: 60                25            RTS
                        26  ;
                        27  AKKU      DFS 1

** ENDE **
```

Wie Sie sehen, haben wir auch hier wieder einen kleinen Trick angewandt. Nach dem Shift laden wir immer eine „0" in den Akkumulator. Dadurch wird das Carrybit nicht verändert, so daß wir erst danach schauen können, ob diese Entscheidung richtig war. Mit BCC verzweigen wir dann nach OUTPUT. War die Entscheidung falsch, so ist das Carrybit gesetzt und wir überschreiben die „0" im Akkumulator mit einer „1", die dann ausgegeben wird. Würden wir nach ASL sofort das Carrybit testen, entstünde ein kompliziertes „Umeinanderherumspringen":

```
        ...
        ASL AKKU
        BCS EINS    ;Bit war 1
        LDA #"0"    ;Bit war 0
        JMP OUTPUT  ;zur Ausgabe
EINS    LDA #"1"
OUTPUT  JSR COUT    ;ausgeben
        ....
```

Nach dem LDA #"0" müssen wir den folgenden Befehl LDA #"1" überspringen. Wir können das mit einem JMP (Sprung) nach OUTPUT tun. Auch ein BCC OUTPUT führt zum Ziel, da das Carrybit an dieser Stelle immer gelöscht sein muß, weil sonst schon vorher eine Verzweigung bei BCS stattgefunden hätte. Für Besitzer des 65C02 gibt es eine Alternative: BRA OUTPUT. BRA steht für BRanch Always, Verzweige immer. Hier wird keine Bedingung getestet, sondern immer verzweigt. Die Adressierung ist wie bei den anderen Branches relativ, also maximal 127 Bytes vor oder 128 Bytes zurück. Mit einem Jump können Sie dagegen beliebig weit vor- oder zurückspringen. BRA braucht aber nur 2 Bytes im Maschinencode (JMP 3 Bytes). In der Geschwindigkeit unterscheiden sich beide nicht.

Wenn Sie es ganz trickreich mögen: es geht sogar noch kürzer als im obigen Programm:

```
        ...
        ASL AKKU
        LDA #%01011000 ; $58
        ROL
        JSR COUT
        ...
```

Durch den Befehl ROL ergibt sich folgendes:
a) Carry gelöscht 01011000 ---> 10110000 = $B0 = „0"
b) Carry gesetzt 01011000 ---> 10110001 = $B1 = „1"
So erhalten wir ganz automatisch das richtige ASCII-Zeichen.

Lektion 26: Ein wenig Logik

Der 6502-Prozessor hat die Möglichkeit, den Inhalt von Speicherplätzen mit dem Akkumulatorinhalt logisch zu verknüpfen. Er beherrscht dabei die UND (AND), ODER (OR) sowie EXCLUSIV-ODER (EOR) Verknüpfungen. In den folgenden Wahrheitstabellen können Sie ihre Bedeutung ersehen:

	0	1
0	0	0
1	0	1

AND

Abb. 17: AND

	0	1
0	0	1
1	1	1

OR

Abb. 18: OR

	0	1
0	0	1
1	1	0

EOR

Abb. 19: EOR

Das Mnemonic zur UND-Verknüpfung heißt AND. AND ist ein Bit-für-Bit-Befehl. Jedes einzelne Bit des Akkumulators wird mit seinem korrespondierenden Bit der gewählten Speicherstelle verglichen. Sind beide gesetzt, wird eine 1 in den Akkumulator eingetragen. Ist nur eines gesetzt oder gar keines, so wird eine 0 eingetragen.
AND wird meistens als Maske benutzt, um bestimmte Bits aus einem Byte **aus**zublenden. Das betreffende Byte wird in den Akkumulator geholt und dann in der unmittelbaren Adressierung mit einem Maskenbyte UND-verknüpft. Im Maskenbyte sind die Bits auf 1 gesetzt, die im Akkumulator stehen bleiben sollen. Jede 0 im Maskenbyte bewirkt, daß auch das betreffende Bit im Akkumulator auf 0 gesetzt wird. Sehr häufig kommt es vor, daß das Bit 7 gelöscht werden soll:

```
LDA WERT         ;Wert laden
AND #%01111111   ;Maske $7F
```

Bitte beachten Sie, daß ein AND #$7F aus einer vorzeichenbehafteten Ganzzahl *keinen* absoluten Wert macht. Dazu ist die Umwandlung in das Zweierkomplement notwendig.
Da AND auch die Null-Flagge setzt, wenn das Ergebnis der Verknüpfung Null ist, eignet sich der Befehl gut dazu, einzelne Bits zu testen:

```
LDA WERT        ;Wert laden
AND #$00100000  ;Maske $20
BEQ ISTNULL     ;Bit 5 gelöscht
```

BEQ verzweigt im obigen Beispiel, wenn Bit 5 von WERT nicht gesetzt war, da dann die UND-Verknüpfung den Wert $00 im Akku zurückläßt.
Ein AND #$FF (= %11111111) ist sinnlos, da dadurch nichts im Akkumulator verändert wird.

Das Mnemonic zur ODER-Verknüpfung heißt ORA. Auch ORA ist ein Bit-für-Bit-Befehl. Jedes einzelne Bit des Akkumulators wird mit seinem korrespondierenden Bit der gewählten Speicherstelle verglichen. Ist eines von beiden oder sind beide gesetzt, wird eine 1 in den Akkumulator eingetragen. Ist keines gesetzt, so wird eine 0 eingetragen.
ORA wird meistens als Maske benutzt, um bestimmte Bits in ein Byte **ein**zublenden. Das betreffende Byte wird in den Akkumulator geholt und dann in der unmittelbaren Adressierung mit einem Maskenbyte ODER-verknüpft. Im Maskenbyte sind die Bits auf 1 gesetzt, die auch im Akkumulator gesetzt werden sollen. Sehr häufig muß das Bit 7 gesetzt werden:

```
LDA WERT        ;Wert laden
ORA #%10000000  ;Maske $80
```

Diese Operation kommt deshalb in vielen Programmen vor, weil die Routine COUT für die Ausgabe normaler Zeichen verlangt, daß Bit 7 gesetzt ist. Bei gelöschtem Bit 7 erhalten wir blinkende oder inverse Zeichen.

Die Verknüpfung ORA #$00 ist wirkungslos bis auf das Setzen der Null- und Negativ-Flagge entsprechend dem (unveränderten) Akkumulator.

Unsere dritte und letzte Verknüpfung ist die Exlusiv-ODER-Funktion mit dem Mnemonic EOR. Alle gesetzten Bits im OPERANDEN **invertieren** die korrespondierenden Bits im Akkumulator: aus einer 1 wird eine 0 und aus einer 0 eine 1. Im OPERANDEN nicht gesetzte Bits bewirken keine Änderung.
EOR #%11110000 bewirkt, daß die Bits 4 bis 7 invertiert werden, während die Bits 0 bis 3 unverändert bleiben. Mit EOR #$11111111 können wir das Einerkomplement einer Zahl bilden. Wir erhalten daraus das Zweierkomplement, wenn wir noch zusätzlich eine Eins addieren. Wird zweimal mit dem selben Wert EOR-verknüpft, hebt sich die Wirkung auf. Dies dient zu einfachen Verschlüsselungstechniken.

Viel benutzt wird EOR im Zusammenhang mit der hochauflösenden Grafik. Es ist damit möglich, zwei Bilder (z.B. HGR1 und HGR2) zu überlagern, ohne den Inhalt der Bilder zu zerstören.

Während die Befehle AND, ORA und EOR nur den Inhalt des Akkumulators beeinflussen, besitzt der 65C02 auch zwei logische Befehle, die die Inhalte von Speicherstellen direkt verändern und den Akku unverändert lassen.
TRB (Test and Reset Bits, Teste Bits und lösche sie) verknüpft die mit dem OPERANDEN bezeichnete Speicherstelle bitweise UND mit dem Akkumulator und invertiert sie anschließend. Dadurch werden im adressierten Byte alle Bits gelöscht, die im Akkumulator gesetzt sind. Die gesamte Befehlsfolge lautet immer:

```
LDA MASKE      ;Bitmuster laden
TRB SPEICHER
```

Ein Beispiel: das Byte SPEICHER enthält das Bitmuster %00001111 (= $0F). Der Akkumulator wird mit $10101010 (= $AA) geladen.

```
LDA #$AA       %10101010
TRB SPEICHER   %00001111
--------------------
    SPEICHER   %00000101 = $05
```

TSB (Test and Set Bits, Teste Bits und setze sie) verknüpft die mit dem OPERANDEN bezeichnete Speicherstelle bitweise ODER mit dem Akkumulator. Dadurch werden im adressierten Byte alle Bits gesetzt, die auch im Akkumulator gesetzt sind. Auch hier lautet die gesamte Befehlsfolge:

```
LDA MASKE      ;Bitmuster laden
TSB SPEICHER
```

Wir gehen von den selben Bitmustern wie im obigen Beispiel aus:

```
LDA #$AA       %10101010
TSB SPEICHER   %00001111
--------------------
    SPEICHER   %10101111
```

Wir haben damit fast alle Instruktionen des 6502/65C02 kennengelernt, die auf der Bitebene arbeiten. In der nächsten Lektion kommt noch ein etwas exotischer Befehl hinzu.

Lektion 27: BIT für Bits

Ein halber Zwilling des AND ist der Befehl BIT. BIT steht für „test BITs" (teste Bits) und führt ein logisches UND zwischen dem im OPERANDEN bezeichneten Speicherbyte und dem Akkumulator durch und setzt dabei die Null-Flagge genau wie der AND-Befehl. Hier enden die Gemeinsamkeiten aber auch schon. BIT verändert **nicht** den Wert des Akkumulators. Zusätzlich werden aber die Bits 7 und 6 des getesteten Bytes in die Bits 7 und 6 des Statusregisters übertragen, also in das Negativbit (N) und die Überlauf-Flagge (V). Mit den beiden Befehlen BPL (Branch on PLus, Verzweige bei gelöschter Negativ-Flagge) und BMI (Branch on MInus, Verzweige bei gesetzter Negativ-Flagge) können wir dann die N-Flagge testen. Für die V-Flagge gibt es die beiden bedingten Verzweigungen BVS (Branch on oVerflow Set, Verzweige bei gesetzter Überlauf-Flagge) und BVC (Branch on oVerflow Clear, Verzweige bei gelöschter Überlauf-Flagge).

Wenn Sie nur Bit 7 oder 6 einer Speicherstelle testen wollen, können Sie das jederzeit mit BIT tun, auch wenn der Akkumulator und die Indexregister „besetzt" sind:

```
BIT SPEICHER                    BIT SPEICHER
BMI NEGATIV  ;Bit 7 gesetzt     BVS GESETZT   ;Bit 6 gesetzt
```

Wenn Sie beispielsweise auf einen beliebigen Tastendruck warten wollen, aber kein Register verlieren dürfen, schreiben Sie einfach folgenden Code:

```
          ....
SCHLEIFE BIT $C000     ;Tastatur
         BPL SCHLEIFE  ;nicht gedrückt
         BIT $C010     ;Tastendruck löschen (Strobe)
         ....
```

Mit Ausnahme des Statusregisters wird hier nichts verändert.

Wenn Sie den Akkumulator als Maske einsetzen, können Sie mit BIT testen, ob mindestens eines von bestimmten Bits in einem Byte gesetzt ist.

```
LDA #$03        ; %00000011  Maske
BIT SPEICHER
BNE JA          ; mindestens ein gesetztes
                  Bit stimmt überein
```

Wenn Sie allerdings testen wollen, ob alle interessierenden Bits gleichzeitig gesetzt sind, kommen Sie um ein AND nicht herum:

```
LDA SPEICHER
AND #$03        ; %00000011  Maske
CMP #$03        ; vollständige Übereinstimmung?
BEQ GLEICH      ; Ja, alle Maskenbits gesetzt
```

Beim 65C02-Prozessor gibt es im Gegensatz zum 6502 auch die mit dem X-Register indizierten Adressierungen und eine unmittelbare Adressierung (z.B. BIT #$80). Bei letzterer werden allerdings nicht Bit 7 und 6 in das Statusregister übertragen, wodurch die Anwendungsmöglichkeiten etwas eingeschränkt sind.

Lektion 28: Erst voll, aber schon übergelaufen

In der letzten Lektion haben wir die Überlauf-Flagge im Zusammenhang mit dem BIT-Befehl schon erwähnt. Ihre eigentliche Funktion ist aber bisher ungeklärt.
Wenn wir zwei Bytes addieren und die Summe paßt nicht mehr in ein Byte, dann wird die Übertrags-Flagge gesetzt (Carry). Wenn wir zwei Bytes subtrahieren und die Differenz ist kleiner als Null, dann wird das Carrybit gelöscht. Bei vorzeichenbehafteten Zahlen stehen nur 7 Bits im Byte für den Zahlenwert zur Verfügung, da das Bit 7 das Vorzeichen angibt. Wenn bei der Addition oder Subtraktion zweier vorzeichenbehafteter Zahlen das Ergebnis nicht mehr in die 7 Bits paßt, dann tritt ein arithmetischer Überlauf (Overflow) ein. 80 + 70 = 150 paßt durchaus noch in ein Byte und erzeugt damit kein gesetztes Carrybit. Der Wertebereich eines (vorzeichenbehafteten) Bytes reicht aber nur von -128 bis +127, und 150 ist größer als 127. Umgekehrt gibt -5 + 7 = 2 ein gesetztes Carrybit, da -5 identisch ist mit dem nichtvorzeichenbehafteten 251, und 251 + 7 ist 258 und damit größer als 255.
Um diesen Schwierigkeiten aus dem Wege zu gehen, besitzt der 6502 die Überlauf-Flagge (V-Flagge) im Bit 6 des Statusregisters. Die Befehle ADC und SBC setzen und löschen diese Flagge, wenn ein Überlauf stattfindet bzw. wenn nicht.
Wenn wir bei einer Addition oder Subtraktion auf die einzelnen Bits schauen, so sehen wir, daß ein Übertrag bzw. ein Borgen an jeder einzelnen Stelle möglich ist. Wir können also auch im Innern eines Bytes so etwas wie ein internes Carrybit sehen. Nur der Übertrag von Bit 7 landet (außen) in der Übertrags-Flagge des Statusregisters. Wenn wir den (internen) Übertrag von Bit 6 nach Bit 7 und den (externen) Übertrag von Bit 7 ins Carrybit Exclusiv-ODER verknüpfen, dann gibt das Resultat die Stellung der Überlauf-Flagge an:

a) intern 1 extern 1 ---> V-Flagge 0
b) intern 1 extern 0 ---> V-Flagge 1
c) intern 0 extern 1 ---> V-Flagge 1
d) intern 0 extern 0 ---> V-Flagge 0

Die Befehlsfolge für eine Addition zweier vorzeichenbehafteter Ganzzahlen ist damit folgende:

```
  LDA ZAHL1
  CLC
  ADC ZAHL2
  BVS ÜBERLAUF  ;außerhalb -128 bis +127
```

Bitte beachten Sie, daß die Vergleichsbefehle CMP, CPX und CPY die V-Flagge nicht beeinflussen, obwohl sie intern mit einer Subtraktion arbeiten. Vorzeichenbehaftete Zahlen können deshalb nicht auf einfache Weise miteinander verglichen werden. Stattdessen ist eine ganze Routine notwendig. Um festzustellen, ob Zahl1 kleiner ist als Zahl2 (beide vorzeichenbehaftet), können wir schreiben:

```
     LDA ZAHL1      ;Differenz bilden
     SEC
     SBC ZAHL2      ;und N- und V-Flagge setzen
     BVS OVER       ;Überlauf
     BPL NEIN       ;nicht kleiner
     BMI JA         ;kleiner
OVER BPL JA         ;kleiner
NEIN ...            ;nicht kleiner
```

Wenn ein Überlauf auftritt, kehrt sich die Logik der N-Flagge um. Sie sollten mit dem Simulator IDUS einmal ein paar Zahlen zur eigenen Übung untersuchen.

Mit CLV (CLear oVerflow, lösche die Überlauf-Flagge) können Sie die V-Flagge per Befehl wieder löschen. Dies wird in der Praxis wenig benutzt und dient auf 6502-Systemen meistens nur dazu, eine definierte, gelöschte V-Flagge zu erhalten, um danach mit BVC verzweigen zu können. Mit einem 65C02 ist BRA kürzer.

```
  CLV        ;V-Flagge löschen
  BVC ZIEL   ;immer verzweigen
```

Lektion 29: Tief im Keller sitz' ich drin

Die Speicherseite $01 (Adressen $0100 bis $01FF) hat beim 6502/65C02 eine besondere Bedeutung. Sie wird vom Prozessor als zeitweiliger Zwischenspeicher für Register oder für Rückkehradressen eines Unterprogrammaufrufs oder nach Hardware- oder Softwareunterbrechungen (die wir erst in Lektion 31 besprechen) benutzt. Die Information wird in der selben Weise gespeichert wie wir Teller im Schrank stapeln: der zuletzt auf den Stapel gelegte Teller wird als erster wieder heruntergenommen. Einen solchen Stapel nennen wir einen LIFO-Speicher (last-in, first-out = zuletzt hinein, zuerst heraus). Der Stapel des 6502, der englisch Stack heißt, hat noch eine besondere Eigenschaft: er steht auf dem Kopf. Das heißt, das erste Byte wird bei $01FF zwischengespeichert, das zweite Byte bei $01FE, das dritte Byte bei $01FD usw. Unser Stapel wächst also nach unten.
Der 6502-Prozessor besitzt ein 8-Bit Register, den Stapelzeiger (Stackpointer), der auf den nächsten freien Platz im Stapel zeigt. Wenn ein Byte auf den Stapel geschoben (pushed) wird, wird dieses Register automatisch um 1 erniedrigt. Wenn ein Byte vom Stapel gezogen (pulled) wird, wird der Stapelzeiger um 1 erhöht. Da alle Adressen des Stack die Form $01XX besitzen, enthält der Stapelzeiger nur das niederwertige Byte der Adresse und der 6502 ergänzt selbsttätig das konstante höherwertige Byte $01.
Normalerweise brauchen Sie sich um den Stackpointer nicht zu kümmern, da er vom Prozessor verwaltet wird. Sie müssen allerdings ein paar Regeln beachten, wenn Sie selbst Stackoperationen vollführen wollen:

a) Jedes auf den Stack geschobene Byte muß auch wieder von dort heruntergezogen werden.
b) Wenn Sie mehrere Bytes hintereinander auf den Stack schieben, werden sie in der umgekehrten Reihenfolge wieder heruntergezogen.
c) Sie sollten den Stackpointer nur nach reiflicher Überlegung selbst verändern.

Diese Regeln sind bis jetzt nur Theorie. Schauen wir uns deshalb an, wozu wir den Stack benutzen können. Der 6502 besitzt zwei Befehle, um Register auf den Stack zu schieben und zwei, um sie wieder herunterzuholen:

PHA PusH Accumulator (schiebe den Akkumulator auf den Stack)
PLA PULl Accumulator (ziehe den Akkumulator vom Stack)

PHP PusH Prozessor status (schiebe das Statusregister auf den Stack)
PLP PuLl Prozessor status (ziehe das Statusregister vom Stack)

Abgesehen von der Tatsache, das sie verschiedene Register betreffen, arbeiten PHA und PHP gleich: der Inhalt des Registers wird an der mit dem Stackpointer bezeichneten Speicherstelle abgelegt, und dann wird der Stackpointer um 1 dekrementiert, so daß er auf die nächst niedrigere Adresse zeigt.
Ganz ähnlich erhöhen PLA und PLP zuerst den Stackpointer um 1 und laden dann das betreffende Register mit der bezeichneten Speicherstelle, die dadurch wieder frei wird.
Der 65C02 besitzt noch vier weitere Stackbefehle, um auch die Indexregister auf den Stapel schieben bzw. sie von dort wieder herunterziehen zu können.

PHX PusH X-Register (schiebe das X-Register auf den Stack)
PLX PuLl X-Register (ziehe das X-Register vom Stack)

PHY PusH Y-Register (schiebe das Y-Register auf den Stack)
PLY PuLl Y-Register (ziehe das Y-Register vom Stack)

Diese Befehle werden vor allen Dingen benutzt, um kurzfristig ein oder mehrere Register zu sichern. Dies wird besonders häufig vor oder nach dem Aufruf von Unterprogrammen, die diese Register verändern könnten, getan. Der Vorteil liegt darin, daß Sie nicht extra eine Speicherstelle im Hauptspeicher für diesen Zweck definieren müssen. PHA/PLA ist außerdem einen Prozessortakt schneller und 4 Bytes im Code kürzer als STA SPEICHER/LDA SPEICHER.
Mit einem 65C02 können Sie den Akku und die Indexregister folgendermaßen „retten":

```
PHA
PHX
PHY
...  Rest der Routine
PLY
PLX
PLA
```

Wenn wir A, X und Y auf den Stapel schieben, steht Y an der letzten Position. Aus diesem Grund wird Y auch als erstes wieder heruntergezogen, dann X und zuletzt der Akkumulator.
Mit dem 6502 geht das alles etwas umständlicher, da wir hier nur die Befehle für den Akkumulator nutzen können. Wir müssen deshalb die Indexregister erst in

den Akkumulator transportieren, bevor wir sie auf den Stapel schieben können, und nach dem Herunterziehen vom Akku in das Indexregister zurücktransferieren.

```
PHA   ;A retten
TXA   ;X nach A
PHA   ;X retten
TYA   ;Y nach A
PHA   ;Y retten
...   ;Rest der Routine
PLA   ;Y zurück
TAY   ;A nach Y
PLA   ;X zurück
TAX   ;A nach X
PLA   ;A zurück
```

Da wir die Transfer-Befehle bisher nur am Rande gestreift haben, folgt hier noch einmal eine Zusammenstellung:

TAX Transfer Accumulator to X-register
TAY Transfer Accumulator to Y-register
TXA Transfer X-register to Accumulator
TYA Transfer Y-register to Accumulator
TSX Transfer Stackpointer to X-register
TXS Transfer X-register to Stackpointer

Die beiden letzten Befehle sind der einzige Weg, den Stapelzeiger direkt zu versetzen. TXS wird im allgemeinen benutzt, um ein RESET (Zurücksetzen) für den Stack zu erreichen:

```
LDX #$FF
TXS       ;Stapelzeiger auf $01FF
```

Dabei gehen natürlich sämtliche Informationen auf dem Stack verloren. In mindestens 99% aller Assemblerprogramme besteht nicht die Notwendigkeit, als Benutzer in die Stackverwaltung einzugreifen.
Zum Abschluß noch eine kleine Anmerkung: Bei manchen deutschen Autoren heißt der Stack auch Keller oder Kellerspeicher, in den ein- und ausgekellert wird, als handele es sich um Briketts oder Kartoffeln.

Lektion 30: Wo komme ich her?

In der Lektion 19 haben wir schon den Aufruf von Unterprogrammen mittels JSR und die Rückkehr durch RTS benutzt. Jetzt ist es an der Zeit zu erklären, wie der 6502 es eigentlich handhabt, immer zur richtigen Adresse zurückzukommmen.
Ein Unterprogrammaufruf läuft wie folgt ab: durch JSR $HHLL wird zunächst die später benötigte Rücksprungadresse-1 gebildet (die -1 hat ihren Grund in der Hardware des Prozessors), damit der Prozessor beim Rücksprung auf den nächsten Befehl trifft. Im Programmzähler (Program-Counter) merkt sich der Prozessor immer die Speicherstelle des gerade abzuarbeitenden Befehls. Die daraus berechnete Rücksprungadresse wird auf den Stack geschoben (zuerst das Hi-Byte und dann das Lo-Byte) und der Stackpointer um 2 erniedrigt. Dann

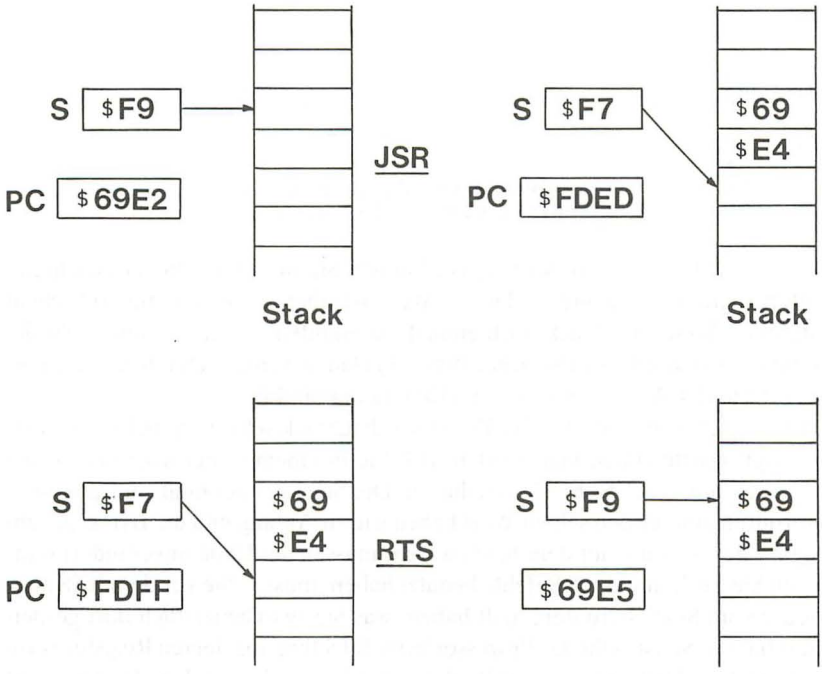


Abb. 20: JSR und RTS

wird die gewünschte Sprungadresse in den Programmzähler geladen und die Abarbeitung dort fortgesetzt.
Der Rücksprungbefehl RTS holt die zwischengespeicherte Rücksprungadresse wieder vom Stack, erhöht sie um den Wert 1 und läd sie dann in den Programmzähler. Dadurch erhält dieser die Adresse des Befehls, der unmittelbar auf das JSR folgt. Der Stackpointer wird natürlich wieder um 2 heraufgesetzt.
Sie können sich merken: JSR „pushed" 2 Bytes, RTS „pulled" 2 Bytes.

Wenn Sie mehrere Subroutinen ineinander verschachteln, dann werden die Rücksprungadressen nacheinander auf den Stack geschoben. Jedes RTS zieht dann immer die Rücksprungadresse des letzten JSR vom Stack. Damit ist die richtige Reihenfolge gewährleistet.
In BASIC gibt es den Befehl POP, der eine Rücksprungadresse löscht. Auch in der Assemblersprache können Sie leicht einen Rücksprung auslassen: Sie ziehen einfach die entsprechende Adresse mit zwei PLA vom Stack:

```
      ....
EINS  JSR ZWEI
      ....
      ....
ZWEI  JSR DREI
      ....
DREI  ....
      PLA
      PLA         ; 1 Adresse vom Stack ziehen
      RTS         ; springt hinter EINS zurück
```

Da der Stack nur 256 Bytes lang ist, können Sie maximal 128 verschachtelte Unterprogramme benutzen. Dieser Wert ist aber mehr von theoretischem Interesse. Wenn der Stack doch einmal „überlaufen" sollte, besteht nicht die Gefahr, daß irgendwo im Speicher Bytes abgeladen werden. Durch das konstante Hi-Byte der Adresse folgt nach $0100 wieder $01FF.
Da Sie gesehen haben, daß der Prozessor den Stack selbsttätig bei jedem JSR benötigt, dürfte Ihnen klar werden, daß Sie in einem Unterprogramm keine Bytes auf dem Stack stehen lassen dürfen. Der Stackpointer muß am Ende einer Subroutine immer den selben Wert haben wie zu Anfang und die Bytes, die am Beginn der Routine auf dem Stack waren, müssen am Ende unverändert sein. Wenn Sie Push- und Pull-Befehle benutzt haben, müssen Sie vor dem RTS alles wieder vom Stack heruntergeholt haben, was Sie zwischenzeitlich dort gespeichert haben. Sonst zieht der Prozessor beim RTS Ihre abgelegten Registerwerte vom Stack und behandelt sie als Rücksprungadresse. Wo Sie dann landen, steht in den Sternen.

Die Tatsache, daß ein RTS immer die beiden letzten Bytes vom Stack zieht, um dann das Programm an der so gefundenen Adresse (+1) fortzusetzen, läßt sich für einen kleinen Trick ausnutzen. Mit zwei PHA schieben Sie eine von Ihnen gewünschte Adresse auf den Stack und führen dann ein RTS aus. Bei diesem RETURN eines vorgetäuschten JSR wird das Programm an der von Ihnen beabsichtigten Stelle fortgesetzt. Da RTS zwei Bytes vom Stack zieht, ist damit die Stellung des Stackpointers wieder neutralisiert.
Wozu soll das gut sein, wenn Sie mit einem Jump doch auch überall hinkommen? Nun, oft kann es sein, daß sich die Sprungadresse erst im laufenden Programm ergibt. Dann können Sie keinen festen JMP einbauen. Menüs sind ein häufiger Ort für den eben beschriebenen Trick. Schreiben wir eine kleine Routine, die auf einen Tastendruck wartet. Bei „A", „S", „D" und „F" wird jeweils zu einer anderen Stelle gesprungen, alle anderen Tastendrucke werden nicht angenommen. Sie können selbstverständlich andere Zeichen wählen und auch ihre Anzahl verringern oder vergrößern. Nur dürfen es theoretisch nicht mehr als 127 sein, aber soviele unterschiedliche Möglichkeiten hat sicherlich auch Ihre Tastatur nicht.

```
                        1    ;*****************
                        2    ; Menü-Verteiler *
                        3    ;*****************
                        4    ;
                        5    ; Liest ein Zeichen von der Tastatur
                        6    ; und verzweigt je nach Tastendruck
                        7    ;
                        8    ; Definitionen
                        9    ;
                        10   KEY       EQU $C000
                        11   STROBE    EQU $C010
                        12   BELL      EQU $FBDD
                        13   ;
                        14             ORG $1000
                        15   ;
                        16   ; Programm
                        17   ;
1000: AD 00 C0          18   MENUE     LDA KEY
1003: 10 FB             19             BPL MENUE       ;Taste?
1005: 8D 10 C0          20             STA STROBE      ;Ja
1008: A2 04             21             LDX #$04        ;4 Möglichkeiten
100A: CA                22   SCHLEIFE  DEX             ;nächster Buchstabe
100B: 30 F3             23             BMI MENUE       ;neu versuchen
100D: DD 1E 10          24             CMP CODE,X      ;vergleichen
1010: D0 F8             25             BNE SCHLEIFE    ;nicht gleich
1012: 8A                26             TXA             ;Index bilden
1013: 0A                27             ASL             ;* 2
1014: AA                28             TAX             ;zurück für Index
1015: BD 23 10          29             LDA SPRUNG+1,X  ;Zieladresse
```

```
1018: 48          30              PHA             ;Hi-Byte auf Stack
1019: BD 22 10    31              LDA SPRUNG,X
101C: 48          32              PHA             ;Lo-Byte auf Stack
101D: 60          33              RTS             ;anspringen
                  34    ;
101E: C1 D3 C4    35    CODE      ASC "ASDF"
                  36    ;
1022: 29 10       37    SPRUNG    ADR ZIEL1-1     ;1. Ziel
1024: 2A 10       38              ADR ZIEL2-1     ;2. Ziel
1026: 2B 10       39              ADR ZIEL3-1     ;3. Ziel
1028: 2C 10       40              ADR ZIEL4-1     ;4. Ziel
                  41    ;
                  42    ; hier nur Pseudoziele
                  43    ;
102A: EA          44    ZIEL1     NOP
102B: EA          45    ZIEL2     NOP
102C: EA          46    ZIEL3     NOP
102D: 4C DD FB    47    ZIEL4     JMP BELL

** ENDE **
```

In der Schleife wird der im Akkumulator gespeicherte Tastendruck mit den 4 Wahlmöglichkeiten verglichen. Das X-Register dient dabei als Schleifenzähler. Wenn X noch einmal dekrementiert wird, nachdem es den Wert $00 enthält, klappt das Byte zu $FF um. Das ist per Definition eine negative Zahl, so daß die N-Flagge im Statusregister gesetzt wird. Sie zeigt damit an, daß keine der 4 Möglichkeiten mit dem Tastendruck identisch war. Durch BMI (Branch on MInus, Verzweige bei gesetzter Negativ-Flagge) kehren wir deshalb zum Programmanfang zurück. Sie können diesen Sprung auch erst zu einer Routine lenken, die auf den Fehler aufmerksam macht (im einfachsten Falle ein „Piep" im Lautsprecher), und dann zu MENUE zurückkehren. Wurde der Tastendruck in der Tabelle CODE gefunden, wird das X-Register mit 2 multipliziert. Dies geschieht einfach durch eine Linksverschiebung. Da dieser Befehl für X nicht existiert, wird X erst in den Akkumulator transferiert (TXA), dann dort multipliziert und wieder nach X zurückgeholt (TAX). Diese Multiplikation ist notwendig, da die Zieladressen alle 2 Bytes lang sind. Ihr relativer Start gemessen an SPRUNG ist:

1. Adresse 0 Bytes Offset
2. Adresse 2 Bytes Offset
3. Adresse 4 Bytes Offset
4. Adresse 6 Bytes Offset.

Da die Ziele normalerweise alle ein LABEL besitzen, können wir einen weiteren Pseudo-Opcode des Assemblers benutzen, um die Sprungtabelle zu bauen. ADR steht für Adresse. In zwei aufeinanderfolgenden Bytes wird die Adresse des OPERANDEN im Lo/Hi-Format abgelegt. Bei den Adressenan-

gaben ist zu berücksichtigen, daß durch RTS eine 1 addiert wird. Wir müssen also in die Tabelle die um 1 verminderten Zieladressen eintragen. Dann werden nacheinander erst das Hi- und dann das Lo-Byte auf den Stack gebracht. Da wir X nicht verändern wollen und das Hi-Byte im Speicher hinter dem Lo-Byte liegt, besorgt auch hier der Assembler die nötige Addition von 1 zur Bezugsadresse (Zeile 29).

An den Zieladressen bauen Sie dann natürlich Ihre Routinen ein. In diesem Beispiel sollte eigentlich nichts anderes bewirkt werden, als anzuzeigen, daß wir angekommen sind. Deshalb wird die Ihnen schon aus Lektion 7 bekannte Monitor-Routine BELL für ein akustisches Signal benutzt. Wir hätten nun natürlich für jedes Ziel JMP BELL schreiben können. Einfacher aber war es, nur einmal BELL zu benutzen und von allen Zielen bis in die Zeile 47 „durchzufallen". Der Befehl NOP steht für „No OPeration", was soviel heißt wie „Tue nichts". Der Prozessor verbraucht hier lediglich zwei Takte und geht dann zum nächsten Befehl über. NOP wird teilweise benutzt, um Codeteile für Test zu überschreiben (3 NOPs können ein JSR $HHLL ersetzen, wodurch das Unterprogramm übergangen wird). In anderen Fällen werden die 2 Takte bei zeitkritischen Routinen für einen Feinabgleich benutzt. Im DOS stehen z.B. viele NOPs, da es beim Schreiben auf die Diskette auf jede Mikrosekunde ankommt.

Lektion 31: Eine kleine Unterbrechung

Es gibt drei verschiedene Möglichkeiten, der Ablauf eines Programms zu unterbrechen (wenn wir vom Abschalten des Stroms einmal absehen). Eine davon haben Sie sicher schon oft benutzt: RESET. Ein „wildgewordenes" Programm läßt sich damit zumeist stoppen. Wenn Sie die RESET-Taste drükken, so lösen Sie einen sogenannten Hardware-Break aus. Es gibt von der Taste eine direkte Verbindung zum Prozessor. Wenn das Signal am Prozessor ankommt, springt er **immer** zu der Adresse, die in den Speicherstellen $FFFC und $FFFD abgelegt ist. Da sich diese im ROM befinden, ist der Beginn des RESET-Zyklus erst einmal festgelegt. Bei den verschiedenen Monitoren gibt es allerdings verschiedene Zieladressen:

Alter Monitor: $FF59
Autostart/IIe/IIc: $FA62

Der ganz alte Apple II springt nach ein paar Initialisierungen zur selben Adresse, die Sie auch mit CALL-151 erreichen. Auch bei den neueren Versionen finden zunächst einige Initialisierungen statt (Textbildschirm einschalten und löschen usw.) und dann wird das sogenannte POWER-UP-Byte $03F4 getestet. Wenn es nicht den Wert von $03F3 EOR #$A5 besitzt, geht der Apple davon aus, daß er gerade eingeschaltet wurde und vollführt einen Kaltstart, d.h. er versucht von der Diskette zu booten. Andernfalls springt er nach $03F2. Dort wiederum steht normalerweise ein Sprung in den Monitor zurück, der uns das bekannte Sternchen präsentiert. Da $03F2 im RAM steht, können dort auch andere Adressen eingetragen werden, so daß die endgültige Antwort des Apple auf ein RESET durch Software kontrolliert werden kann. Kopiergeschützte Programme setzen meistens einen Rücksprung in sich selbst, damit Sie mit RESET nicht „heraus" kommen.

Wenn Sie bei $03F2 nachsehen, werden Sie dort keinen Jump-Befehl finden, sondern nur die nackte Adresse ähnlich wie in unserer Adresstabelle in der letzten Lektion. Sie ist allerdings nicht um 1 vermindert. Der Monitor kann also den RTS-Trick nicht benutzen, sondern muß eine andere Möglichkeit haben, eine Zieladresse zu lesen. Schauen Sie sich im Benutzer-Handbuch das Assembler-Listing des Monitors einmal an. Sie finden an der Adresse $FAA3 den gesuchten Sprung JMP (SOFTEV).

Dies ist ein Sprung mit **absoluter indirekter Adressierung**. In der Klammer wird die Basisadresse angegeben, die irgendwo im Speicher liegen kann. Aus dieser

holt sich der Prozessor das niederwertige Adressbyte der effektiven Zieladresse. Aus Basisadresse+1 kommt danach das höherwertige Byte.
Ein RESET ist in den meisten Fällen keine Unterbrechung, sondern ein echter Abbruch, da das Programm später nicht wieder aufgenommen wird.
Daneben gibt es zwei echte Unterbrechungen, nach denen das Programm an der alten Stelle wieder aufgenommen wird. Der 6502 hat einen Anschluß mit der Bezeichnung IRQ (Interrupt ReQuest, Unterbrechungsanfrage), der es einem externen Gerät (z.B. einer Zusatzkarte) ermöglicht, dem Prozessor anzumelden, daß es jetzt vom Prozessor „bedient" werden will. Sie können das mit einem klingelnden Telefon vergleichen. Das Klingeln zeigt an, daß irgendjemand etwas von Ihnen will. Genau so, wie Sie jetzt entweder das Klingeln „überhören" oder aber Ihre bisherige Tätigkeit unterbrechen und mit dem Anrufer in Kontakt treten können, so kann der Prozessor auf das IRQ-Signal reagieren oder nicht. Der 6502 ignoriert den IRQ, wenn das Interrupt-disable-bit (Unterbrechungssperre, I-Flagge) im Statusregister auf 1 gesetzt ist, und reagiert auf die Anforderung, wenn es auf 0 steht. Die Flagge kann durch zwei Befehle beeinflußt werden:
CLI CLear Interrupt disable, Lösche die Unterbrechungssperre
SEI SEt Interrupt disable, Setze die Unterbrechungssperre.
Wenn die I-Flagge gelöscht ist und ein IRQ-Signal beim Prozessor eintrifft, findet immer eine vollständige Interrupt-Routine statt:

a) Der augenblicklich bearbeitete Befehl wird noch ausgeführt.
b) Der Programmzähler wird auf den Stack geschoben, das Hi-Byte voran.
c) Das Statusregister wird ebenfalls auf den Stack gebracht, wobei das Break-Bit (B-Flagge, Software-Unterbrechungsflagge) gelöscht ist.
d) Der Programmzähler wird mit der Adresse, die in $FFFE (Lo) und $FFFF (Hi) steht, geladen.
e) Im Statusregister wird die I-Flagge gesetzt, so daß weitere IRQs gesperrt sind.
f) Die Programmausführung wird an der neuen Adresse fortgesetzt.

Diese Adresse ist bei fast allen Monitoren der Apple-Familie verschieden:

Alter Monitor	: $FA86
Autostart/IIe	: $FA40
enhanced IIe	; $C3FA
IIc	: $C803

Nur der IIc und der neue IIe sind mit eigenen Routinen auf eine Bearbeitung von IRQs vorbereitet. Im IIc werden die Maus und die Seriellen Schnittstellen mit Interrupts gesteuert. Es würde an dieser Stelle zu weit führen, diese Routinen zu

beschreiben. In den älteren Apple kommen IRQs normalerweise nicht vor, es sei denn, sie benutzten Zusatzkarten, die damit arbeiten (z.B. alle Uhrenkarten). Deshalb ist in diesen Geräten nur eine rudimentäre Behandlung eingebaut. Die IRQ-Routine in diesen Monitoren springt mit der absoluten indirekten Adressierung zu der Adresse, die in $03FE und $03FF steht. Das ist normalerweise wieder der Monitor-„Eingang" bei $FF65, wo die Routine endet. Eine wirkliche Rückkehr ins laufende Programm ist bei diesen alten Monitoren also nicht vorgesehen. Da aber $03FE und $03FF im RAM liegen, können Sie hier Ihre eigene Interrupt-Routine einbauen.

Zwischen einem normalen Unterprogramm und einer Interrupt-Routine bestehen ein paar prinzipielle Unterschiede. Nach einem normalen Unterprogramm können die einzelnen Register entweder gleich oder verändert sein, je nachdem wie Ihr Programm arbeitet. Manchmal ist es notwendig, die Registerinhalte zu retten, in anderen Fällen nicht. Da ein Unterprogramm immer an einer definierten Stelle im Hauptprogramm aufgerufen wird, lassen sich alle Bedingungen kontrollieren. Ein IRQ kann jedoch jederzeit eintreffen. Deshalb müssen alle Register, auch das Statusregister, am Ende der Interrupt-Routine unverändert sein. Das Statusregister wird vom Prozessor selbst gerettet, um die anderen Register müssen Sie sich kümmern. Die Interrupt-Routine erhält einen neuen Return-Befehl: RTI ReTurn from Interrupt, Kehre zurück von der Interrupt-Routine. RTI zieht zunächst das alte Statusregister vom Stack (wodurch gleichzeitig auch die I-Flagge wieder gelöscht wird) und dann die Rücksprungadresse. Trotzdem ist RTI nicht identisch mit PLP/RTS, da durch den IRQ die exakte Rücksprungadresse (und nicht die Rücksprungadresse-1 wie beim JSR) auf den Stack gebracht wurde. RTI addiert deshalb nicht eine 1 zu der Adresse vom Stack, wie es RTS tut. Auf der Begleitdiskette finden Sie zwei kleine Demonstrationen für den Simulator IDUS, die Ihnen die Wirkungsweise von RTS und RTI veranschaulichen sollen.

Der 6502-Prozessor hat einen weiteren Interrupt-Anschluß, den NMI (Non Maskable Interrupt, nicht maskierbarer Interrupt). Dieser arbeitet ganz ähnlich wie der IRQ, nur daß er mit SEI **nicht** unterdrückt werden kann. Der NMI hat also eine höhere Priorität als der IRQ. Treffen beide gleichzeitig ein, wird der NMI bearbeitet. Bei einem NMI holt sich der Prozessor die neue Adresse aus den Speicherstellen $FFFA (Lo) und $FFFB (Hi), die bei allen Monitoren nach $03FB zeigt. Dort steht wiederum ein Sprung nach $FF65 (JMP $FF65). Sie können hier natürlich Ihre eigene Adresse eintragen. Der Hobby-Programmierer wird allerdings wahrscheinlich nie mit einem NMI konfrontiert werden.

Einen Programmabbruch können Sie auch per Software mit einem Software-

Break erreichen. Dieser wird ganz normal in den Maschinencode eingebaut und hat das Mnemonic BRK (= BReaK, Unterbrechung). BRK kann über die I-Flagge nicht blockiert werden. Wenn der Prozessor auf den BRK-Befehl stößt, verhält er sich wie bei einem IRQ. Nur wird die Unterbrechungs-Flagge (B) im Statusregister diesmal gesetzt. Daran wird in der IRQ-Routine ein BRK von einem IRQ unterschieden. Bei einem Break werden anschließend die Register auf dem Textbildschirm ausgegeben, so als hätten Sie <CTRL-E> im Monitor eingegeben. Die angezeigte Adresse ist im übrigen die Adresse des BRK +2 ! Schließlich „landen" Sie dann auch hier im Monitor mit seinem bekannten Sternchen-Prompt.
BRK eignet sich dazu, in der Testphase eines Programms an bestimmten Stellen Stops einzubauen, um sich dann die Inhalte der Register ansehen zu können. Man nennt dies Break-Point-Debugging (Fehlersuche mit Softwareunterbrechungen). Mit einem Simulator wie IDUS geht die Fehlersuche allerdings wesentlich bequemer, da Sie ständig alle Register „im Griff" haben.
Wenn der 6502 alle Unterbrechungsanforderungen gleichzeitig erhält, setzt er folgende Prioritäten:
1) RESET
2) NMI
3) BREAK
4) IRQ

Lektion 32: Restbestände

Wir haben jetzt fast alle Befehle des 6502 und des 65C02 kennengelernt. Lediglich drei stehen noch aus, mit denen wir uns in dieser Lektion befassen werden.
Der Befehl STZ (STore Zero, Setze auf Null) existiert nur im 65C02. Mit STZ wird eine Speicherstelle auf Null gesetzt. Die Register des Prozessors bleiben dabei unverändert. Beim 6502 müssen Sie dagegen ein Register „opfern", um dasselbe zu erreichen:

```
65C02                6502

STZ SPEICHER         LDA #$00        ;Register initialisieren
                     STA SPEICHER
```

Über die beiden letzten Befehle kann man entweder einige Kapitel schreiben, oder es ganz kurz machen. Ich habe mich für letzteres entschieden.
Wir haben ein Byte bisher als Ganzzahl oder als vorzeichenbehaftete Ganzzahl kennengelernt. Daneben gibt es noch eine dritte Darstellungsweise, die binär codierten Dezimalzahlen (Binary-Coded-Decimal, BCD). Sie reichen von 00 bis 99 in einem Byte. Eine BCD-Zahl besitzt immer 2 Stellen pro Byte, wobei jeweils 4 Bits zu einer Ziffer gehören. 01001000 ist die BCD-Zahl 48, da 0100 gleich 4 und 1000 gleich 8 ist. Bis zu einem gewissen Grade verhalten sich BCD-Zahlen wie HEX-Zahlen: LDA #$15 läd die HEX-Zahl $15 in den Akku. Binär entspricht das %00010101. Wie Sie leicht sehen können, ist das auch die BCD-Zahl 15. BCD-Konstanten können also wie HEX-Zahlen geschrieben werden. Dabei ist allerdings der Wertebereich zu beachten: $00 ist der kleinste und $99 der höchste erlaubte Wert. $FA oder $1D sind **nicht** erlaubt.
Wenn wir BCD-Zahlen addieren oder subtrahieren, benutzen wir dazu auch die Befehle ADC und SBC. Hier gibt es allerdings eine Schwierigkeit. Wenn wir $15 und $15 addieren, erhalten wir $2A. Wir hätten aber gerne $30 als BCD-Zahl gesehen. Die Lösung liegt darin, den Prozessor von der binären Rechenweise auf die dezimale umzuschalten. Dies passiert, indem die Dezimal-Flagge im Statusregister gesetzt (dezimal) oder gelöscht (binär) wird. Wir besitzen dazu zwei Befehle:

CLD Clear Decimal flag, Lösche die Dezimal-Flagge
SED SEt Decimal flag, Setze die Dezimal-Flagge.

Normalerweise sollte die D-Flagge immer gelöscht sein. Der 65C02 setzt sie beim Einschalten und bei jedem RESET genau so. Bei einem 6502 ist die Flagge dagegen undefiniert. Es ist notwendig, in einem Programm erst einmal „klare Verhältnisse" zu schaffen und die Flagge zu löschen. Deshalb fangen die meisten Assemblerprogramme mit CLD an.

Wenn Sie in den Dezimalmodus umgeschaltet haben, produzieren ADC und SBC aus BCD-Zahlen auch BCD-Ergebnisse. Die Inkrement-, Dekrement-, Shift- und Rotationsbefehle arbeiten aber weiterhin rein binär! Mit ASL verdoppelt sich also eine BCD-Zahl nicht. Mit 4 LSRs können Sie allerdings eine BCD-Zahl durch 10 dividieren. Wir müssen also völlig neue Rechenregeln aufstellen.

Das Carrybit arbeitet ganz normal, aber die N-, V- und Z-Flagge sind beim 6502 ungültig. Nur der 65C02 ist hier wieder eine Nasenlänge weiter: Negativ-, Überlauf- und Null-Flagge stimmen auch bei BCD-Zahlen. Für diese zusätzliche Arbeit braucht der Prozessor dann aber auch einen Takt länger.

Es bleibt die Frage, wozu wir eigentlich den Dezimalmodus haben. Ich muß eingestehen, daß ich ihn noch nie ernsthaft benutzt habe, und so wird es sicherlich auch den meisten anderen ergehen. Eine Anwendungsmöglichkeit ist die Speicherung von langen Dezimalzahlen. 666666 benötigt als Dezimalzahl, in ASCII-Form abgespeichert, 6 Bytes (B6 B6 B6 B6 B6 B6), als BCD-Zahl dagegen nur 3 Bytes (66 66 66). Wenn Sie viele lange Zahlen bearbeiten müssen, ist das gepackte dezimale Format platzsparender. Wenn Sie diese Zahlen dann addieren oder subtrahieren müssen, bearbeiten Sie auch jedesmal gleich 2 Ziffern auf einmal, was einen gehörigen Zeitgewinn mit sich bringt. Wenn die mathematischen Aufgaben allerdings anspruchsvoller werden, wird es mit BCD-Zahlen eher schwieriger.

Schmackhafter kann ich Ihnen BCD-Zahlen nicht machen, denn ich habe sie ja selbst, wie schon gesagt, noch nie ernsthaft benutzt.

Lektion 33: Flaggenparade

Unsere Informationen über das Statusregister liegen über viele Lektionen verstreut. Deshalb wollen wir sie hier noch einmal kurz zusammenfassen.
Das Statusregister des 6502 ist in seiner Bedeutung gänzlich verschieden von den anderen Registern. Sein Inhalt interessiert uns nicht als Byte, sondern jedes einzelne Bit hat für sich genommen seine Bedeutung. Vier dieser Bits werden als Ergebnis eines gerade ausgeführten Befehls gesetzt. Zwei weitere werden durch spezielle 6502-Befehle beeinflußt und ein Bit zeigt an, ob die letzte Unterbrechung (Interrupt) durch die „BRK"-Instruktion hervorgerufen wurde. Das Bit 5 schließlich wird nicht benutzt und ist immer auf 1 gesetzt.
Wenden wir uns jetzt etwas ausführlicher diesen Bits zu, die auch als Flaggen bezeichnet werden. Der Begriff „Flagge" ist dabei natürlich nur bildlich zu verstehen: hat das Bit den Wert 1, bedeutet das eine gesetzte Flagge, ist das Bit gleich 0, so ist die Flagge gelöscht (eingeholt).

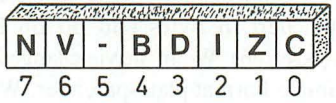


Abb. 21: Statusregister

Bit 0 des Statusregisters ist die **Carry-** oder **Übertrags-Flagge**. Sie wird gesetzt, wenn bei einer Addition, Rotation oder Verschiebung ein Übertrag auftritt. Entsteht bei einer Subtraktion ein Übertrag (Borgen), so wird das Carrybit gelöscht. Auch Vergleichsoperationen können das Carrybit setzen oder löschen. Außerdem gibt es zwei Befehle, die direkt das Carrybit beeinflussen: CLC (= CLear Carry) löscht und SEC (= SEt Carry) setzt die Übertrags-Flagge.

Bit 1 ist die **Zero-** oder **Null-Flagge**. Sie wird gesetzt, wenn der Akkumulator, ein Indexregister oder eine Speicherstelle den Wert $00 als Ergebnis einer arithmetischen oder logischen Operation, eines Rotations-, Verschiebungs-, Inkrement- oder Dekrementbefehls oder durch Laden enthält. Diese Flagge kann nicht direkt durch einen Befehl beeinflußt werden.

Bit 2 ist die **Interrupt-Disable-Flag** oder **Unterbrechungssperre**. Sie wird gesetzt, wenn ein Interrupt-Request (Unterbrechungsanforderung) an dem IRQ-

Pin des Prozessors angekommen ist, und sperrt damit weitere Unterbrechungen. Durch die Ausführung des abschließenden RTI-Befehls (ReTurn from Interrupt) wird sie wieder gelöscht. Mittels SEI (= SEt Interrupt disable) bzw. CLI (= CLear Interrupt disable) kann sie auch per Befehl gesetzt und gelöscht werden.

Bit 3 ist die **Dezimal-Flagge**. Sie kann nur durch den Befehl SED gesetzt bzw. durch CLD gelöscht werden. Beim 65C02 löscht ferner ein <RESET> die Dezimal-Flagge. Sie teilt dem Prozessor mit, ob bei Additionen und Subtraktionen die beteiligten Zahlen binär (Flagge gelöscht) oder dezimal (Flagge gesetzt) codiert sind.

Bit 4 ist die **Break-** oder **Software-Unterbrechungs-Flagge**. Sie ist normalerweise gesetzt und bleibt es auch, wenn ein BRK-Befehl (= Break) ausgeführt wird. Auch die I-Flagge wird in diesem Fall gesetzt. Bei einem IRQ (Interrupt-

N	1	Negativ	
	0	Positiv	
V	1	Überlauf	
	0	kein Überlauf	
-			
B	1	Break	
	0	IRQ	
D	1	Dezimal	
	0	Binär	
I	1	IRQ-Sperre	Ein
	0		Aus
Z	1	Null	
	0	nicht Null	
C	1	Übertrag	
	0	kein Übertrag	

Abb. 22: Bedeutung der Bits

Request) wird die B-Flagge dagegen gelöscht und die I-Flagge gesetzt. Es gibt keinen direkten Befehl, um diese Flagge zu verändern.

Bit 5 ist undefiniert und immer gesetzt.

Bit 6 ist die **Overflow-** oder **Überlauf-Flagge**. Sie wird gesetzt, wenn bei einer Addition oder Subtraktion ein Ergebnis entsteht, das größer als $7F (127) oder kleiner als $80 (-128) ist. Dies ist nur sinnvoll bei vorzeichenbehafteten Zahlen. Mit dem Befehl CLV (= CLear oVerflow) kann die V-Flagge auch per Befehl gelöscht werden. Auch die BIT-Instruktion beeinflußt die V-Flagge.

Bit 7 ist die **Negativ-Flagge**. Sie wird gesetzt, wenn Bit 7 des Akkumulators, eines Indexregisters oder einer Speicherstelle den logischen Wert 1 auf Grund einer Operation des Microprozessors erhält. Es gibt keinen direkten Befehl zur Veränderung dieser Flagge.

Mit den bedingten Verzweigungen werden die Flaggen getestet. Im Verlauf eines Programms ändert sich die Stellung der Flaggen ständig. Deshalb muß ein Flaggentest in der Regel immer unmittelbar nach dem Befehl stehen, dessen Auswirkung auf die Flaggen getestet werden soll.

Lektion 34: A weiß, daß B weiß, wo C wohnt

Der 6502 hat 13 verschiedene Adressierungsarten. Die meisten haben wir schon kennengelernt, ein paar „vertraxte" fehlen uns noch. Versuchen Sie nicht, alle Möglichkeiten auswendig zu lernen. Fassen Sie diese Lektion als einen Überblick auf, in dem Sie immer wieder nachschlagen können. Die Details lernen Sie am besten durch die Praxis.

Die einfachste Adressierung ist die **implizite Adressierung** (implied), weil bei ihr in nur einem Byte alle Informationen enthalten sind. Wir haben Beispiele dafür kennengelernt. TAY besagt z.B., daß der Inhalt des Akkumulators in das Y-Register transferiert werden soll. Aber auch RTS enthält in einem Byte alle benötigten Informationen, ebenso alle Befehle, die Flaggen des Statusregisters direkt verändern (CLC, SEI, ...).

Die **Akkumulator-Adressierung** benötigt ebenso nur 1 Byte und bezieht sich immer auf den Akkumulator. Der 6502 kennt nur 4 solcher Befehle: ASL, LSR, ROL und ROR. Beim 65C02 kommen noch INA und DEA hinzu.

Die **unmittelbare Adressierung** (immediate) benötigt zwei Bytes und bezieht sich immer auf das zweite Byte, das eine Konstante enthält. Wollen Sie den Akkumulator z.B. mit dem Wert $80 laden, so lautet der Assemblerbefehl LDA #$80 und der daraus resultierende Maschinencode A9 80. Die „80" steht unmittelbar hinter dem Befehlsbyte „A9". Der Assembler erkennt diese Adressierungsart an dem Nummernkreuz #. Da $80 eine HEX-Zahl ist, darf auch das Dollarzeichen nicht fehlen.

Die **absolute Adressierung** benötigt drei Bytes, wobei die letzten beiden eine Adresse im Apple-Speicher bezeichnen. Im Maschinencode lautet „Lade den Akkumulator mit dem Inhalt der Speicherstelle $10D0" wie folgt:

AD D0 10.

Beim 6502-Prozessor ist es üblich, bei einer Adresse zuerst den niederwertigen Teil (hier $D0) zu nennen und dann das höherwertige Adressbyte (hier $10), also zuerst die relative Position und dann die Speicherseite. Mit einem Assembler wie ASSESSOR brauchen Sie diese „Verdrehung" bei der Programmeingabe nicht zu machen, das wird für Sie erledigt. Der Befehl lautet hier: LDA $10D0.

Mit den beiden Bytes der Adresse lassen sich 65536 Speicherstellen ($0000 bis $FFFF) adressieren, also der gesamte 64K Hauptspeicher des Apple.

Obwohl in der Maschinensprache das Befehlsbyte für die unmittelbare und für die absolute Adressierung unterschiedlich ist (A9 bzw. AD), benutzen Sie im Assembler für die gleiche Operation (hier Akkumulator laden) immer das selbe Mnemonic, ganz unabhängig von der Adressierungsart. Die passende Übersetzung besorgt der Assembler, wenn Sie ihn durch die Art der Adresse richtig instruieren.

Eine Abart der absoluten Adressierung ist die **Nullseiten-Adressierung**, die wiederum nur 2 Bytes benötigt. Mit ihr können nur die ersten 256 Bytes von $0000 bis $00FF bearbeitet werden. Sie können sie als absolute Adressierung auffassen, deren höherwertiges Byte konstant $00 ist und daher nicht angegeben werden muß.

Maschinencode:	A5	E2	85	00
Assemblercode:	LDA	$E2	STA	$00

Sie geben also nur ein Byte der Adresse ein. Die meisten Assembler sind intelligent genug, auch in LDA $00E2 eine Nullseiten-Adressierung zu erkennen und zu übersetzen. Die Nullseiten-Adressierung kann zu kompakterem Code und schnellerem Programmablauf führen, wenn sie mit Bedacht eingesetzt wird.

Die **relative Adressierung** (zwei Bytes lang) wird nur mit den Verzweigungsbefehlen (Branch) verwendet. Das zweite Byte des Befehls stellt einen Offset (Distanzbyte) dar, der zu dem Inhalt des Programmzählers (Program Pointer) addiert wird. Der Offset liegt im Bereich von -128 und +127. Mit einem Assembler brauchen Sie den Offset nicht selber auszurechnen. Sie geben einfach die Zieladresse ein, den Rest besorgt der Assembler.

Kommen wir jetzt zu den etwas komplizierteren, aber auch reizvolleren Möglichkeiten, den indizierten Adressierungen. Da ist zunächst die **absolute indizierte Adressierung**, die 3 Bytes benötigt. Als Maschinencode finden wir z.B.
BD 03 08 und in Assembler LDA $0803,X.

Das bedeutet, daß der Akkumulator mit dem Wert der Speicherstelle geladen werden soll, die die Adresse $0803 **plus** den Inhalt des X-Registers hat. Das wäre $0808, falls im X-Register ein $05 steht.

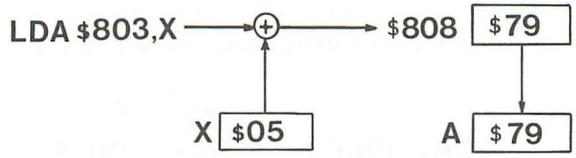


Abb. 23: absolute indizierte Adressierung

Mit dem Y-Register funktioniert alles genauso, nur daß Sie hier ein „,Y" an die Basisadresse anhängen. Wollen Sie das X- oder Y-Register selbst laden, so ist natürlich nur das andere Register als Index zugelassen:

```
LDX $1050,Y     (BE 50 10)
LDY $1050,X     (BC 50 10)
```

Die gleichwertigen Speicherbefehle existieren leider nicht.

Auch von den entsprechenden Formen der **indizierten Nullseiten-Adressierung** gibt es nicht alle Varianten. Es sind nur

```
LDA $35,X       (B5 35)
LDX $35,Y       (B6 35)
LDY $35,X       (B4 35)
```

sowie

```
STA $35,X       (95 35)
STX $35,Y       (96 35)
STY $35,X       (94 35)
```

zulässig.

Die effektive Adresse wird errechnet, indem das Indexregister zu der Nullseiten-Basisadresse addiert wird. Das Ergebnis ist immer auch eine Adresse auf der Seite $00, da bei einer indizierten Nullseiten-Adressierung *kein* Übertrag in das höherwertige Adressbyte stattfindet. LDA $FE,X mit X = $05 adressiert also $0003 und nicht etwa $0103!

Nun machen wir es noch etwas raffinierter. Nehmen wir zunächst wieder ein umgangssprachliches Beispiel: „Gehe zum Fremdenverkehrsamt und laß Dir dort die Adresse der Apotheke sagen, hinter der sich im fünften Haus ein Lederwarengeschäft befindet." Diese Variante, die in Assemblerprogrammen sehr häufig vorkommt, heißt **indirekte indizierte Adressierung** und lautet z.B.

```
LDA ($1E),Y     (B1 1E).
```

Die Basisadresse **muß** auf der Nullseite liegen, als Indexregister ist nur das Y-Register zugelassen.

Wenn der Prozessor auf diesen Befehl stößt, holt er zunächst das Byte, das in der angegebenen Basisadresse (hier $001E) steht und betrachtet es als den nieder-

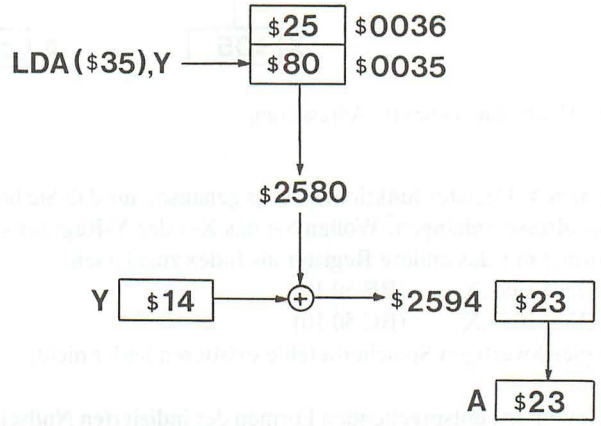


Abb. 24: indirekte indizierte Adressierung

wertigen Teil der Zieladresse. Danach holt er das Byte aus der folgenden Speicherposition (hier $1F) und betrachtet es als den höherwertigen Anteil (indirekter Anteil). Schließlich wird noch der Inhalt des Y-Registers addiert (indizierter Anteil), um zur effektiven Adresse zu gelangen.
Nehmen wir einmal an, daß sich in $001E und $001F die Werte $43 und $20 befänden, und daß das Y-Register mit $15 geladen sei. Als effektive Adresse erhielten wir dann:

```
  $2043
+$0015
-------
  $2058
```

Zum Laden oder Speichern der Indexregister existiert diese Adressierung nicht.

Eher selten benutzt wird für den Akkumulator die **indizierte indirekte Adressierung**. Umgangssprachlich lautet sie: „Fünf Häuser hinter dem Fremdenverkehrsamt wohnt jemand, der die Adresse des gesuchten Geschäftes kennt". Hierfür eignet sich nur das X-Register und der Assemblercode lautet:

```
LDA ($35,X)      (A1 35)
```

Wieder muß die Basisadresse auf der Nullseite liegen. Beachten Sie, daß hier *zuerst* das Indexregister zur Basisadresse addiert und dann dort die effektive

(endgültige) Adresse gefunden wird. Bei der indirekten indizierten Adressierung wurde dagegen erst die Zieladresse aus der Basisadresse gelesen und *danach* der Index addiert. Aus diesem Grunde heißt die indizierte indirekte Adressierung (LDA ($35,X)) auch vorindizierte indirekte Adressierung und die indirekte indizierte Adressierung (LDA ($15),Y) auch nachindizierte indirekte Adressierung.

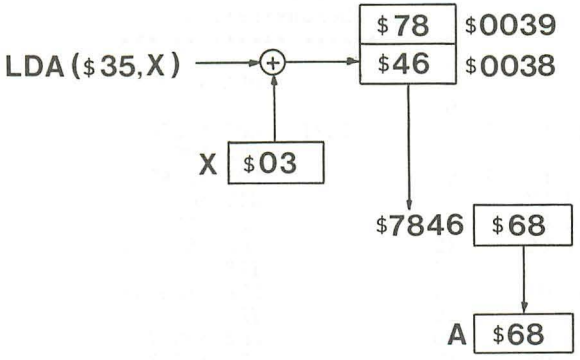


Abb. 25: indizierte indirekte Adressierung

Betrachten wir noch ein Beispiel. Auf der Nullseite befinden sich folgende Werte:
1A: 80 25 43 FE 29 F6.

Mit den X- und Y-Registern gleich $00 erhalten wir:
LDA ($1A),Y --> Adresse $2580
LDA ($1A,X) --> Adresse $2580.
Soweit sind beide Formen gleichwertig.

Mit X- und Y-Register gleich $02 erhalten wir dagegen:
LDA ($1A),Y --> Adresse $2580 + $02 = $2582
LDA ($1A,X) --> Adresse $FE43.

Mit X- und Y-Register gleich $04 schließlich erhalten wir:
LDA ($1A),Y --> Adresse $2580 + $04 = $2584
LDA ($1A,X) --> Adresse $F629.

Falls Sie noch Schwierigkeiten mit den vielen verschiedenen Adressierungsarten haben, machen Sie sich einmal in Ruhe klar, was der Prozessor wann woher

nimmt und zu welchem Zeitpunkt jeweils der Index addiert wird. Der Index wird übrigens immer addiert, so daß nur auf höhere Adressen zugegriffen werden kann. Das Programm „DEMO 3" wurde speziell dafür geschrieben, Ihnen viele Varianten im Simulator IDUS zu zeigen. Etwas Sinnvolles vollbringt dieses Programm sonst nicht.

```
                    1    ;********************
                    2    ;  Demonstration 3  *
                    3    ;********************
                    4    ;
                    5              ORG $300
                    6    ;
0300: AD F6 03      7      START   LDA $03F6
0303: 85 06         8              STA $06
0305: AC F7 03      9              LDY $03F7
0308: 84 07         10             STY $07
030A: A0 00         11             LDY #$00
030C: B1 06         12             LDA ($06),Y
030E: AA            13             TAX
030F: A1 06         14             LDA ($06,X)
0311: A8            15             TAY
0312: B6 06         16             LDX $06,Y
0314: B5 06         17             LDA $06,X
0316: 6C 06 00      18             JMP ($06)

** ENDE **
```

Für den 6502 gibt es nur noch eine weitere Adressierung: die **absolute indirekte Adressierung**, die lediglich für eine Sprunganweisung (JMP) zur Verfügung steht. Sie benötigt immer 3 Bytes, auch wenn die Basisadresse auf der Nullseite liegt. Aus der Basisadresse wird das niederwertige Adressbyte der effektiven Adresse geholt. Der nächste Speicherplatz enthält dann das höherwertige Byte. Der Inhalt der Indexregister X und Y ist ohne Einfluß auf das Ergebnis. Das obige Beispiel enthält in Zeile 18 diese Adressierung.

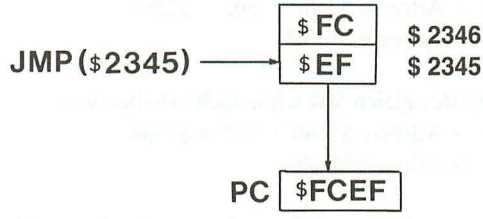


Abb. 26: absolute indirekte Adressierung

Bei dem neuen 65C02 sind noch weitere Adressierungsarten dazugekommen. Die erste ist die **indirekte Nullseiten-Adressierung**, die 2 Bytes benötigt. Das zweite Byte zeigt zu der Nullseiten-Adresse, die das niederwertige Byte der effektiven Adresse enthält. Die folgende Nullseiten-Speicherstelle enthält das höherwertige Byte. Der Inhalt der Indexregister ist ohne Einfluß. Diese Adressierung entspricht in der Wirkung der indirekten indizierten oder indizierten indirekten Adressierung mit dem Wert $00 im Indexregister. Allerdings bleibt hier das Index-Register für andere Zwecke frei.

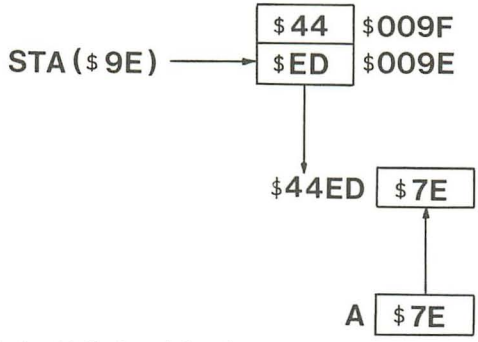


Abb. 27: indirekte Nullseiten-Adressierung

Zu guter Letzt wurde auch das „Repertoire" an Sprüngen noch ergänzt durch die **absolute indizierte indirekte Adressierung**. Hinter diesem Wortungetüm verbirgt sich eine sinnvolle Erweiterung besonders für die Fälle, wenn ein Sprungziel einer Tabelle entnommen werden soll. Es werden immer 3 Bytes benötigt. Zum zweiten und dritten Byte der Anweisung (Lo/Hi-Format) wird der Inhalt des X-Registers addiert. Das Ergebnis der Addition zeigt auf die Speicherstelle, die das niederwertige Byte der effektiven Adresse enthält. Im folgenden Byte findet sich der höherwertige Anteil.

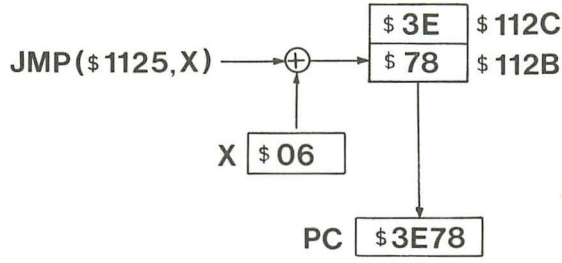


Abb. 28: absolute indizierte indirekte Adressierung

In dieser Lektion wurde schwerverdauliche Kost serviert. Erst mit etwas Übung werden Sie die Kunst der Adressierung beherrschen lernen, das hier gebotene Skelett bekommt Fleisch und Farbe. Im Kapitel 4 mit der vollständigen Beschreibung des Befehlssatzes finden Sie alle erlaubten Adressierungen noch einmal angegeben. Die Übersichtstabelle gibt Ihnen weitere Hilfen.

Lektion 35: Ein Zuhause für mein Programm

Wenn Sie eigene Programme schreiben, müssen Sie festlegen, wo diese im Speicher abgelegt werden sollen. Dazu ist es notwendig, die Speicherbelegung des Apple zu kennen. In Lektion 6 haben wir uns schon einen Überblick verschafft. Wir werden uns jetzt noch einmal etwas ausführlicher mit den verschiedenen „Seiten" des Apple beschäftigen.

Ganz „unten" befindet sich die *Seite $00*, die für Assemblerprogrammierer eine besondere Bedeutung hat, da es für die **„Zero Page"** (= Null-Seite) spezielle Prozessorbefehle gibt. Die eingebauten Programme des Apple machen intensiv von ihr Gebrauch, und wir können sie für unsere eigenen Programme nur nach sorgfältiger Prüfung, ob wir nicht andere Teile des Apple dabei stören, einsetzen.

Es schließt sich die *Seite $01* an, die für uns weitgehend tabu ist, da sie der Prozessor selbst benötigt. Sie trägt den Namen **„Stack"** oder „Stapel". Wir können sie nur für kurzzeitige Zwischenspeicherungen mitbenutzen.

Die *Seite $02* ist der **Tastatureingabepuffer**. In ihr werden unter Applesoft BASIC Tastendrucke bis zu einem <RETURN> zwischengespeichert. Da maximal 256 Zeichen einschließlich des <RETURN> Platz finden, verstehen Sie jetzt auch, warum Sie keine BASIC-Zeilen beliebiger Länge eingeben können. Normalerweise laufen in Seite $02 keine Programme ab, da sie von der Tastatur überschrieben würden.

Die *Seite $03* ist an ihrem oberen Ende ab $03D0 mit sogenannten „Vektoren" fest belegt. Vektoren sind Zeiger auf andere Programme, z.B. auf das Diskettenbetriebssystem. Wir können sie mit Wegweisern vergleichen, die wir uns in eigenen Programmen zunutze machen können.

Zwischen $0300 und $03CF befindet sich freier Platz, ausreichend für kleinere Anwenderprogramme. Die meisten Beispiele dieses Bandes laufen in diesem Bereich.

Die *Seiten $04* bis *$07* haben keinen festen Inhalt, aber eine feste Funktion: sie werden ständig auf dem Bildschirm angezeigt. Als Basic-Programmierer kennen Sie diesen Bereich als Textseite 1 oder LORES 1. Auch hierhin sollten Sie keine Programme legen.

Von *Seite $08* an aufwärts befindet sich freier Programmspeicher. Der Beginn Ihrer Applesoftprogramme wird z.B. nach $0801 geladen. Wenn Sie das Diskettenbetriebssystem DOS benutzen, endet der freie Bereich normalerweise vor

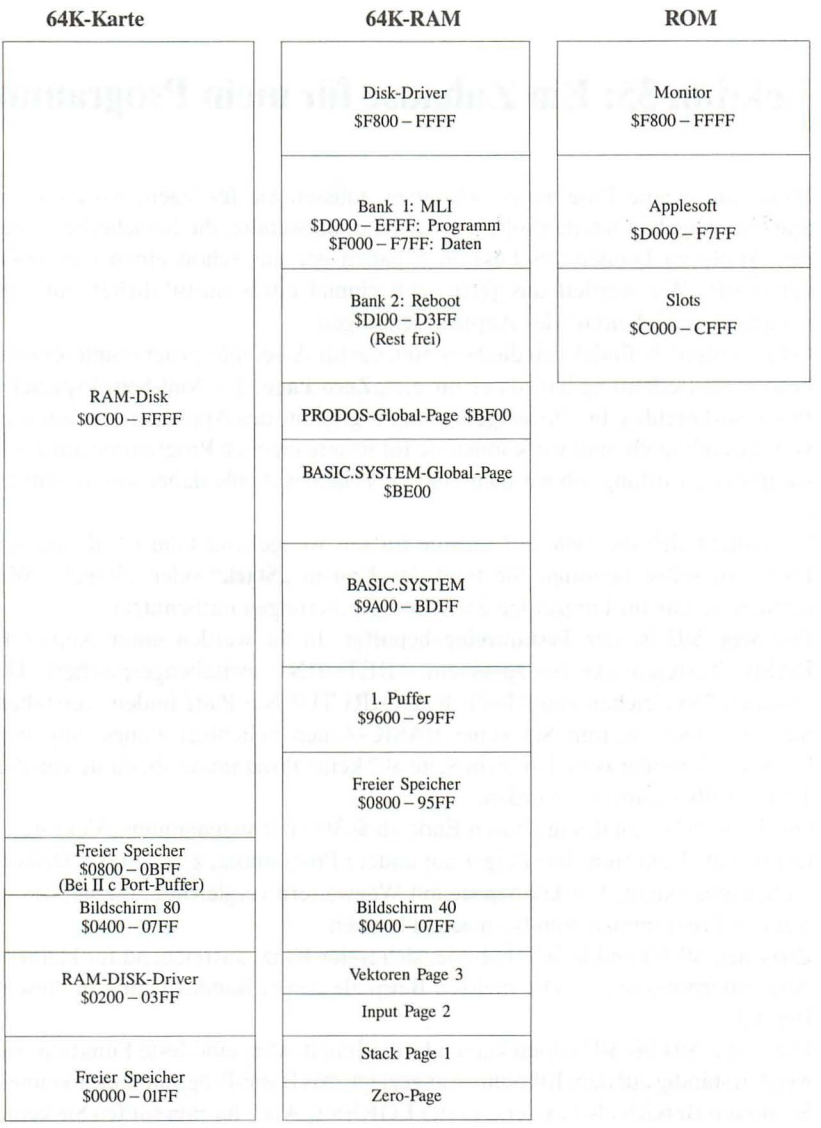


Abb. 29: Speicheraufteilung unter ProDOS

$9600, also mit der Seite $95. Die ganzen ca. 35 KByte stehen Ihnen ungeschmälert nur zu, wenn sie keine hochauflösende Grafik benutzen. HGR1 belegt die *Speicherseiten* $20 bis $3F, HGR2 die *Seiten* $40 bis $5F. Diese Lage mitten im Programmspeicher (die historische Ursachen hat) führt zu vielen Unbequemlichkeiten bei der Entwicklung von langen Programmen, die geschickt um die Grafik herum plaziert werden müssen.

In den *Seiten* $96 *bis* $BF inclusive residiert **DOS 3.3** oder ein Teil von **ProDOS**. Mit den *Seiten* $C0 bis $CF hat es eine ganz besondere Bewandnis. Dort werden nämlich die Ein- und Ausgabeoperationen getätigt. Bestimmte Peripheriegeräte gehören zu festen Adressen, z.B. der Lautsprecher zur Speicherstelle $C030 oder die Tastatur zu $C000.

Alle Seiten bis hierhin, also von $00 bis $CF, bestehen aus sogenanntem **Schreib-Lese-Speicher** (**RAM** = Random Access Memory). Das heißt, Sie können in diesem Bereich sowohl Werte lesen als auch durch eine Schreiboperation Werte verändern. Die verbleibenden Seiten $D0 bis $FF bestehen aus **Nur-Lese-Speicher** (**ROM** = Read Only Memory), der nicht verändert werden kann.

In den *Seiten* $D0 bis $F7 ist der **Applesoft**-Interpreter abgelegt und von $F8 bis $FF folgt der **System-Monitor**. Dieses ist ein Programm, das für alle grundlegenden Operationen des Apple zuständig ist, wie z.B. für das Lesen der Tastatur oder für die Ausgabe von Zeichen auf den Bildschirm.

Wie Sie aus der Abbildung 29 entnehmen können, existiert noch ein Teil des RAM-Hauptspeichers. Es ist die sogenannte Language Card (= Sprachkarte, 16K-Karte). Beim alten Apple IIplus war sie auf einer richtigen Steckkarte vorhanden, bei den heutigen IIe und IIc ist sie direkt auf der Hauptplatine untergebracht und nicht so augenfällig sichtbar. Dieser Bereich ist wieder ein RAM-Speicher und belegt die Seiten $D0 bis $FF. Die Adressen liegen parallel zum ROM des Apple (Applesoft, Monitor), so daß sie zweimal vorhanden sind, einmal als ROM und einmal als RAM. In der Seite $C0 gibt es nun „Umschalter", die entweder den ROM- oder den RAM-Bereich aktivieren. Näheres über die Speicherverwaltung erfahren Sie erst im zweiten Band, denn es existiert eine ganze Handvoll von Möglichkeiten.

Nach dem Einschalten des Apple ist immer der ROM-Bereich aktiv. Der RAM-Bereich bleibt für Programme frei, wenn Sie DOS 3.3 verwenden, oder wird von ProDOS belegt, das ziemlich viel Speicher „frißt". USCD-PASCAL oder andere nachladbare Sprachen benutzen diesen Speicherplatz. Daher kommt der Name „Language Card" oder „Sprachkarte".

Nun werden Sie sich sicher fragen, warum es diese umschaltbaren Speicherbereiche, die „**Bänke**" gibt, und nicht einfach neue Adressen vergeben werden.

Der Microprozessor des Apple besitzt nur 16 Adressleitungen, mit denen genau 2 Byte dargestellt werden können, eben die Adressen von $0000 bis $FFFF. Sollen größere Speicher bearbeitet werden, müssen sie mit Schaltern in diesen Adressbereich umgelegt werden (Bank-Switching), bevor sie angesprochen werden können.
Die Rechner IIe und IIc besitzen zusätzlich noch ROM mit den Adressen von $C100 bis $CFFF, das parallel zum I/O-RAM der Seiten $C1 bis $CF liegt. Auch diese Bereiche werden umgeschaltet.

Der IIc und ein IIe mit erweiterter 80-Zeichenkarte besitzen parallel zum Hauptspeicher noch einmal 64K RAM, die ebenfalls in Bänken umgeschaltet werden müssen und unter ProDOS z.T. als RAM-Disk dienen.

In diesem ersten Band des Assembler-Lehrgangs haben wir noch keine großen Programme geschrieben. Da Sie aber jetzt alle Befehle kennen, können Sie sich daran machen, eigene Vorstellungen zu entwickeln und in Programme umzusetzen. Zu einem vollständigen Programm fehlen uns zwar noch einige spezielle Kenntnisse, aber die werden wir uns gemeinsam im 2. Band aneignen. Dort wird es um formatierte Bildschirmausgaben gehen, um hoch- und niedrigauflösende Grafik, um die Bedienung der Diskettenlaufwerke unter DOS und ProDOS, um die musikalischen Eigenschaften des Apple und vieles mehr. Neben COUT und BELL werden wir dabei lernen, viele ROM-Routinen des Apple sinnvoll zu nutzen. Wir werden dazu den Assembler ASSESSOR benutzen und auch IDUS wird uns weiter begleiten. Von diesem ersten Band werden Sie dann sicher noch oft das Kapitel 4 benötigen, das den vollständigen Befehlssatz der beiden Prozessoren 6502 und 65C02 zusammen mit vielen Programmierbeispielen enthält.
Eine gute Quelle für Programmierideen sind auch Computer-Fachzeitschriften. Die meisten drucken Assemblerlistings ab. Wenn Sie diese eingehend studieren, werden Sie viel dazulernen können. Empfehlen möchte ich Ihnen die Zeitschrift „PEEKER“ aus dem Hüthig-Verlag, die für jeden Schwierigkeitsgrad Maschinenprogramme veröffentlicht. Wenn Sie halbwegs gut Englisch verstehen, ist auch „NIBBLE“ von MicroSPARC Inc. eine zuverlässige Quelle. In Deutschland ist diese amerikanische Zeitschrift in einigen Computerläden oder über den Pandasoft Versand in Berlin erhältlich.

4. Der 6502/65C02 Befehlssatz

Zeichenerklärung:

* = 1 Takt mehr bei Seitenübergang
+ = 2 Takte bei Nicht-Verzweigung, 3 Takte bei Verzweigung, 4 Takte bei Verzweigung mit Seitenübergang
7/6 = 7 Takte bei 6502, 6 Takte bei 65C02
5/6 = 5 Takte bei 6502, 6 Takte bei 65C02

ADC

ADd with Carry
Addiere mit Übertrag zum Akkumulator

Der Wert des mit dem OPERANDEN gewählten Speicherbytes wird zum Akkumulator addiert, und *zusätzlich* wird eine 1 für ein **gesetztes** Carrybit (C-Flagge) addiert. Das Ergebnis findet sich im Akkumulator, das Carrybit erhält den neuen Übertrag. Die Flaggen N, V, Z und C werden entsprechend dem Ergebnis gesetzt.
ADC arbeitet sowohl im Dezimal- als auch im Binärmodus. Die Flaggen sind beim 6502 nur im Binärmodus gültig, beim 65C02 in beiden Modi.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
ADC #.	69	2	2	Unmittelbar
ADC .	65	2	3	Zero-Page
ADC .,X	75	2	4	Zero-Page indiziert mit X
ADC ..	6D	3	4	Absolut
ADC ..,X	7D	3	4*	Absolut indiziert mit X
ADC ..,Y	79	3	4*	Absolut indiziert mit Y
ADC (.,X)	61	2	6	Indiziert indirekt mit X
ADC (.),Y	71	2	5*	Indirekt indiziert mit Y
ADC (.)	72	2	5	Indirekt (65C02)

ADC ist der einzige Additionsbefehl des 6502. Um ohne Übertrag zu addieren, muß vor jeder Rechnung das Carrybit mit CLC gelöscht werden.
Beispiel: WERT1 + WERT2 = WERT3

```
CLC
LDA WERT1lo
ADC WERT2lo
STA WERT3lo
LDA WERT1hi
ADC WERT2hi
STA WERT3hi
```

AND

logical AND with accumulator
Logisches UND mit dem Akkumulator

Dieser Befehl verknüpft den Akkumulator und den Wert des mit dem OPERANDEN gewählten Speicherbytes bitweise durch ein logisches UND. Das Ergebnis findet sich im Akkumulator. Die N- und Z-Flaggen werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
AND #.	29	2	2	Unmittelbar
AND .	25	2	3	Zero-Page
AND .,X	35	2	4	Zero-Page indiziert mit X
AND ..	2D	3	4	Absolut
AND ..,X	3D	3	4*	Absolut indiziert mit X
AND ..,Y	39	3	4*	Absolut indiziert mit Y
AND (.,X)	21	2	6	Indiziert indirekt mit X
AND (.),Y	31	2	5*	Indirekt indiziert mit Y
AND (.)	32	2	5	Indirekt (65C02)

UND bedeutet, daß das Ergebnis nur dann 1 ist, wenn *beide* verknüpften Bits 1 sind, sonst ist das Ergebnis 0.
Der AND-Befehl wird hauptsächlich zum Testen bestimmter Bitmuster und für Maskierungen benutzt.

Wahrheitstabelle:

	0	1
0	0	0
1	0	1

AND

Abb. 30: AND

Beispiel: Löschen des Bit 7 von WERT

```
LDA WERT
AND #$7F ;%0111 1111
STA WERT
```

Testen, ob in WERT die Bits 7,1 und 0 *nicht* gesetzt sind

```
LDA WERT
AND #$82 ;%1000 0011
BEQ JA
```

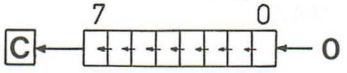
Modulo für Vielfache von 2; WERT1 Modulo 4 = WERT2

```
LDA WERT1
AND #$03 ;%0000 0011
STA WERT2
```

ASL

Arithmetic Shift Left
Arithmetische Verschiebung nach links

Alle Bits des Akkumulators bzw. des bezeichneten Speicherplatzes werden um eine Stelle nach links geschoben. In das frei werdende Bit 0 kommt eine Null, das alte Bit 7 landet im Carrybit (C-Flagge). Die Flaggen N, Z und C werden entsprechend gesetzt.



ASL

Abb. 31: ASL

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
ASL	0A	1	2	Akkumulator
ASL .	06	2	5	Zero-Page
ASL .,X	16	2	6	Zero-Page indiziert mit X
ASL ..	0E	3	6	Absolut
ASL ..,X	1E	3	7/6	Absolut indiziert mit X

ASL wird hauptsächlich für die Multiplikation eines Wertes mit 2 benutzt, da jede Verschiebung nach links den Wert verdoppelt. Es sollte nachher das Carrybit auf Übertrag getestet werden.
Beispiel: Verdoppelung eines 16-Bit Wertes

```
ASL WERTlo
ROL WERThi
BCS Übertrag
```

Fast ebenso häufig ist der Test, ob bestimmte Bits gesetzt sind. (siehe Programm Print-Bits)

BCC

Branch on Carry Clear
Verzweige bei gelöschtem Carrybit

Dieser Befehl ist eine bedingte Verzweigung mit relativer Adressierung. Ist das Carrybit gelöscht, wird verzweigt, andernfalls wird einfach der nächste Befehl ausgeführt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BCC .	90	2	2+	Relativ

BCC steht zumeist unmittelbar nach einem Vergleich, um das Ergebnis zu testen. BCC hat dann dieselbe Bedeutung wie „Verzweige, falls kleiner", weshalb gute Assembler auch den Pseudo-Opcode BLT (Branch on Lower Than) zulassen. Mit CLC kann das Carrybit auch auf Befehl hin gelöscht werden.

Beispiel: Verzweige, wenn Wert1 < Wert2

```
        LDA WERT1
        CMP WERT2
        BCC KLEINER
        BCS GROESSER
```

Ein weiteres Einsatzgebiet ist die Addition eines 8-Bit Wertes WERT1 zu einem 16-Bit Wert WERT2

```
        CLC
        LDA WERT1lo
        ADC WERT2
        STA WERT1lo
        BCC WEITER ;kein Übertrag
        INC WERT1hi
WEITER  ......
```

BCS

Branch on Carry Set
Verzweige bei gesetztem Carrybit

Dieser Befehl ist wie BCC eine bedingte Verzweigung mit relativer Adressierung. Ist das Carrybit gesetzt, wird verzweigt, andernfalls wird einfach der nächste Befehl ausgeführt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BCS .	B0	2	2+	Relativ

BCS steht zumeist unmittelbar nach einem Vergleich, um das Ergebnis zu testen. BCS hat dann dieselbe Bedeutung wie „Verzweige, falls größer oder gleich", weshalb gute Assembler auch den Pseudo-Opcode BGE (Branch on Greater than or Equal) zulassen. Mit SEC kann das Carrybit auch auf Befehl hin gesetzt werden.

Beispiel: Verzweige, falls Wert1 >= Wert2

```
LDA WERT1
CMP WERT2
BCS GROEGLEI
```

BCS wird auch häufig benutzt, um auf Übertrag zu testen (siehe ADC). Einen Befehl für „kleiner oder gleich" gibt es nicht. Sie müssen dann mehrere Tests durchführen:

```
LDA WERT1
CMP WERT2
BCC KLEINER
BEQ GLEICH
BCS GROESSER
```

BEQ

Branch on EQual
Verzweige bei gesetzter Zero-Flagge (Ergebnis Null)

Dieser Befehl stellt eine bedingte Verzweigung mit relativer Adressierung dar. Es wird verzweigt, wenn die Zero-Flagge (Z) gesetzt ist, andernfalls wird einfach der nächste Befehl ausgeführt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BEQ .	F0	2	2+	Relativ

BEQ steht zumeist unmittelbar nach einem Vergleich oder in Schleifen, um zu testen, ob das Ergebnis Null ist.
Beispiel: Schleifenausgang

```
          LDY #$22
SCHLEIFE  LDA TEXT,Y
          BEQ ENDE
          ....
          DEY
          BNE SCHLEIFE
```

BIT

test BITs
Teste Speicherbits gegen den Akkumulator

BIT gehört auf den ersten Blick zu den etwas verwirrenden Befehlen. Der Akkumulator und das im OPERANDEN spezifizierte Speicherbyte werden logisch UND-verknüpft, das resultierende Byte jedoch verworfen! Die Z-Flagge wird aber entsprechend dem Gesamtergebnis gesetzt. Zusätzlich werden die Bits 7 und 6 der getesteten Speicherstelle in die N- bzw. V-Flagge übertragen. Der Akkumulator und die Speicherstelle bleiben unverändert, nur das Statusregister wird beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BIT #.	89	2	2	Unmittelbar (65C02)
BIT .	24	2	3	Zero-Page
BIT .,X	34	2	4	Zero-Page indiziert mit X (65C02)
BIT ..	2C	3	4	Absolut
BIT ..,X	3C	3	4	Absolut indiziert mit X (65C02)

Bei der unmittelbaren Adressierung werden die Bits 6 und 7 **nicht** übertragen. BIT wird überwiegend benutzt, um festzustellen, ob Bit 7 gesetzt ist. Gegenüber einem Vergleich hat BIT den Vorteil, daß der Akkumulator nicht verändert wird.

Beispiel: Warten auf Tastendruck

```
SCHLEIFE BIT TASTE
         BPL SCHLEIFE ;nicht gedrückt
         BIT STROBE
         ....
```

BMI

Branch on MInus
Verzweige, wenn Minus

Die (relativ adressierte) Verzweigung wird ausgeführt, wenn die N-Flagge (Vorzeichen) gesetzt ist. Andernfalls wird der nächste Befehl bearbeitet. Die N-Flagge wird durch eine Operation gesetzt, die ein Ergebnis im Bereich $80 bis $FF liefert (Bit 7 gesetzt), und gelöscht durch ein Ergebnis zwischen $00 und $7F (Bit 7 gelöscht).

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BMI .	30	2	2+	Relativ

BMI wird benutzt, um bei vorzeichenbehafteten Binärzahlen negative Zahlen zu bestimmen. Noch häufiger dient BMI als Test auf ein gesetztes Bit 7, wenn z.B. eine Schleife bis Null heruntergezählt werden soll (Test am Schleifenbeginn).

Beispiel: Zählschleife bis Null

```
         LDY #$1E
SCHLEIFE DEY
         BMI ENDE ;bis Null heruntergezählt
         ....
         ... SCHLEIFE
ENDE     ....
```

BNE

Branch on Not Equal
Verzweige bei gelöschter Zero-Flagge (Ergebnis ungleich Null)

BNE testet die Zero-Flagge (Z) und verzweigt, wenn sie nicht gesetzt ist, also das Ergebis einer Operation von Null verschieden war. Andernfalls wird der nächste Befehl ausgeführt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BNE .	D0	2	2+	Relativ

BNE dient in Schleifen häufig dazu, einen Index bis 1 herunterzuzählen. Auch die Addition oder Subtraktion von 1 von einem 16-Bit Wert kann bequem mit BNE durchgeführt werden.
Beispiel: Zählschleife bis 1

```
            LDY #$0F
SCHLEIFE    LDA TEXT,Y
            ....
            DEY
            BNE SCHLEIFE ;wenn noch größer Null: SCHLEIFE
```

Subtraktion von 1 (16-Bit Wert)

```
            LDA WERTlo
            BNE KEINÜBERTR
            DEC WERThi
KEINÜBERTR  DEC WERTlo
```

Addition von 1 (16-Bit Wert)

```
            INC WERTlo
            BNE WEITER ;kein Übertrag
            INC WERThi
WEITER      ....
```

BPL

Branch on PLus
Verzweige, wenn Plus

BPL testet die N-Flagge (Vorzeichen) und verzweigt, wenn sie nicht gesetzt ist. Andernfalls wird der nächste Befehl ausgeführt. Die N-Flagge wird durch eine Operation gelöscht, die ein Ergebnis im Bereich von $00 bis $7F liefert.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BPL .	10	2	2+	Relativ

BPL wird oft als Test für Bit 7 benutzt. Ein Beispiel finden Sie unter BIT. Auch mit BPL können Sie eine Schleife bis 0 herunterzählen, wobei der Test am Ende der Schleife steht.
Beispiel: Zählschleife bis Null

```
          LDX #$33
SCHLEIFE  ....
          ....
          DEX
          BPL SCHLEIFE
```

BRA

BRanch Always
Verzweige immer

Dieser Befehl stellt eine unbedingte Verzweigung mit relativer Adressierung dar, d.h. es wird immer verzweigt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BRA .	80	2	2+	Relativ (65C02)

Da die Verzweigung nicht von der Stellung irgendeiner Flagge abhängig ist, eignet sich dieser Befehl sehr gut für relocativen Code. Beim 6502 muß vorher eine Flagge extra gesetzt werden, die dann getestet wird und so zur Verzweigung führt. Die Sprungweite ist aber weiterhin auf effektive 127 Bytes vorwärts oder 128 Bytes rückwärts begrenzt
Beispiel: Erzwungene Verzweigung

```
alt:  CLC            neu: BRA ZIEL
      BCC ZIEL
```

BRK

BReaK (software interrupt)
Unterbrechung durch Software

Beginn einer Unterbrechungsabfolge: Der Program Counter wird um zwei erhöht und auf den Stack geschoben (erst Hi-Byte, dann Lo-Byte), die Break-Flagge (B) auf 1 gesetzt und das Statusregister (P-Register) ebenfalls auf den Stack gebracht. Dann werden weitere Interrupts durch Setzen der I-Flagge blockiert. Der Inhalt der Speicherstellen $FFFE (lo) und $FFFF (hi) wird in den Program Counter geladen und ein Sprung zu dieser Adresse durchgeführt. Beim Apple führt er nach $FA40 (bzw. $C803 beim IIc). Dort wird zunächst zwischen Hardware- und Softwareinterrupt unterschieden, die I-Flagge wird wieder gelöscht, die Register gespeichert. Der Autostart-Monitor springt dann zu der Adresse, die in $03F0 und $03F1 abgelegt ist, normalerweise $FA59. Dieser Sprung kann vom Anwender umgelenkt werden. Wenn dies nicht geschehen ist, werden die zwischengespeicherten Register angezeigt und dann der System Monitor angesprungen.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BRK	00	1	7	Implizit

Der BRK-Befehl wird in der Regel nur zu Testzwecken in Programmen verwendet, da er bequem anzeigt, daß der Programmfluß an dieser Stelle angekommen ist, und gleichzeitig der Inhalt aller Register präsentiert wird. Mit einem guten Debugger (wie **IDUS**) kommen Sie allerdings wesentlich schneller ans Ziel. Das Byte $00 wird häufig auch als Endmarkierung in Strings verwendet, dort aber nicht vom Prozessor als Befehl ausgeführt!
Ein gerade eingeschalteter Rechner enthält viele Nullen in seinem RAM-Speicher. Wenn ein Programm „durchdreht", ist die Wahrscheinlichkeit groß, daß es bald auf ein $00 stößt, welches sanft, aber bestimmt den Irrweg beendet.

BVC

Branch on oVerflow Clear
Verzweige bei gelöschter V-Flagge

Wenn die Overflow-Flagge V gelöscht ist, wird verzweigt, andernfalls der nächste Befehl ausgeführt. V kann mit CLV per Befehl gelöscht werden. Wenn bei einer Operation *kein* Übertrag von Bit 6 nach Bit 7 stattfand, ist V ebenfalls gelöscht.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BVC .	50	2	2+	Relativ

BVC dient bei vorzeichenbehafteten Binärzahlen dazu, einen Überlauf in das Vorzeichenbit (Bit 7) zu testen. Dieser Befehl wird selten benutzt. Zusammen mit BIT läßt sich testen, ob Bit 6 einer Speicherstelle gesetzt ist. Zusammen mit CLV läßt sich ein relativer Sprung erzwingen, was besonders für relokativen Code interessant ist.
Beispiel: Test von Bit 6

```
BIT SPEICHER
BVC NICHT ;Bit 6 nicht gesetzt
```

Erzwungene Verzweigung

```
CLV
BVC IMMER
```

BVS

Branch on oVerflow Set
Verzweige bei gesetzter V-Flagge

Wenn die Overflow-Flagge V gesetzt ist, wird verzweigt, andernfalls der nächste Befehl ausgeführt. V kann nicht per Befehl gesetzt werden. Wenn bei einer Operation ein Übertrag von Bit 6 nach Bit 7 stattfand, ist V gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
BVS .	70	2	2+	Relativ

BVS dient bei vorzeichenbehafteten Binärzahlen dazu, einen stattgefundenen Überlauf in das Vorzeichenbit (Bit 7) anzuzeigen. Dieser Befehl wird selten benutzt. Zusammen mit BIT läßt sich testen, ob Bit 6 einer Speicherstelle gesetzt ist.

Beispiel: Test von Bit 6

```
BIT SPEICHER
BVS JA ; Bit 6 gesetzt
```

CLC

CLear Carry
Lösche das Carrybit

Das Carrybit wird auf Null gesetzt. Dies ist u.a. vor einer Addition ohne Übertrag mit ADC notwendig.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
CLC	18	1	2	Implizit

CLC kann zusammen mit BCC auch benutzt werden, um in (relocatiblem) Code eine Verzweigung zu erzwingen.

Beispiel: Addition einer Konstanten zu 1-Byte Werten

```
CLC
LDA WERT
ADC #KONSTANTE
STA WERT
```

Erzwungene Verzweigung

```
CLC
BCC IMMER
```

CLD

CLear Decimal mode
Dezimal-Betriebsart abschalten

CLD löscht die D-Flagge und schaltet den Prozessor vom Dezimalmodus auf den Binärmodus um.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
CLD	D8	1	2	Implizit

Die „übliche" Betriebsart des 6502 ist der Binärmodus. Der Modus beeinflußt ganz wesentlich die Befehle ADC und SBC, die entweder binäre oder BCD-Ergebnisse bringen. Beim 6502 ist der Modus nach dem Einschalten undefiniert, beim 65C02 wird immer auf Binärmodus geschaltet. Alle internen ROM-Routinen des Apple funktionieren nur im Binärmodus einwandfrei. Deshalb ist es eine gute Praxis, jedes Maschinenprogramm mit dem Befehl CLD einzuleiten, um sicher im Binärmodus zu sein.

CLI

CLear Interrupt mask
Lösche die Unterbrechungssperre

CLI löscht die I-Flagge im Prozessorstatus und gibt damit die Unterbrechungsmöglichkeit des 6502 (Hardware Interrupt) frei.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
CLI	58	1	2	Implizit

Der 6502 unterbricht bei einer Interrupt-Anforderung (Interrupt ReQuest IRQ) seine normale Programmausführung und springt in eine im ROM programmierte Routine, die ihrerseits andere Programme starten kann. Nach dem Ende der Interrupt-Behandlung (RTI) wird das ursprüngliche Programm wieder ab der Unterbrechungsstelle aufgenommen. Normalerweise kommen im Apple II+ und IIe keine IRQ's vor, da das Signal auf einer speziellen Leitung erfolgen muß, die nur mit den Slots (Steckerleisten) verbunden ist. Eine Hardware-Uhr oder andere Zusatzeinrichtungen lösen jedoch häufig Interrupts aus. Beim Apple IIc wird beispielsweise die Maus mit Interrupts gesteuert.

Die I-Flagge wirkt als Sperrzeichen: ist sie gesetzt, sind Unterbrechungen nicht gestattet, wird sie gelöscht (z.B. mit CLI), wird die Interruptmöglichkeit wieder freigegeben.

CLV

CLear oVerflow flag
Lösche die Überlaufflagge

CLV löscht die V-Flagge im Prozessorstatus.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
CLV	B8	1	2	Implizit

Für diesen Befehl besteht keine Notwendigkeit, da die Stellung der Überlaufflagge V in keine Rechenoperation des 6502 direkt eingeht. ADC, BIT, CLV, PLP, RTI und SBC setzen oder löschen die V-Flagge als Indikator, ohne vom vorherigen Status beeinflußt zu werden. Lediglich zur Erzwingung von Verzweigungen hat CLV eine praktische Bedeutung.
Beispiel: Erzwungene Verzweigung

```
CLV
BVC IMMER
```

CMP

CoMPare with accumulator
Vergleiche mit dem Akkumulator

Der Akkumulator wird mit dem im OPERANDEN spezifizierten Wert verglichen und die Flaggen N, Z und C entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
CMP #.	C9	2	2	Unmittelbar
CMP .	C5	2	3	Zero-Page
CMP .,X	D5	2	4	Zero-Page indiziert mit X
CMP ..	CD	3	4	Absolut
CMP ..,X	DD	3	4*	Absolut indiziert mit X
CMP ..,Y	D9	3	4*	Absolut indiziert mit Y
CMP (.,X)	C1	2	6	Indiziert indirekt mit X
CMP (.),Y	D1	2	5*	Indirekt indiziert mit Y
CMP (.)	D2	2	5	Indirekt (65C02)

Intern wird der im OPERANDEN adressierte Wert vom Akkumulator abgezogen, der resultierende Wert jedoch verworfen und nur die Flaggen entsprechend gesetzt. Der Akkumulator und der Vergleichswert bleiben unverändert. Wenn der Akkumulator kleiner ist als der Vergleichswert, liegt das Ergebnis unter Null, also im Negativen, wodurch die N-Flagge gesetzt wird (dies gilt nur für vorzeichenbehaftete Zahlen!). Da bei der Subtraktion dann ein Übertrag (Borgen) stattfindet, wird das Carrybit gelöscht.

	Akku < Wert	Akku = Wert	Akku > Wert
N:	(1)	0	(0)
Z:	0	1	0
C:	0	1	1

Üblicherweise steht nach dem Vergleich ein Verzweigungsbefehl. Durch geeignete Wahl (e.v. Kombination) sind alle Bedingungen testbar.
Beispiel: Größentest WERT1 gegen WERT2

```
LDA WERT1
CMP WERT2
BCC KLEINER  ;= BLT
BEQ GLEICH
BCS GROESSER ;= BGE, aber auf „gleich" schon getestet
```

Wenn Sie mit BCC oder BCS ein Ergebnis testen wollen, sollten Sie nie mit CMP #$00 vergleichen, da dann keine echte Entscheidung möglich ist (jeder Wert ist >= 00, setzt also das Carrybit).

CPX

ComPare with X-register
Vergleiche mit dem X-Register

Das X-Register wird mit dem im OPERANDEN spezifizierten Wert verglichen und die Flaggen N, Z und C entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
CPX #.	E0	2	2	Unmittelbar
CPX .	E4	2	3	Zero-Page
CPX ..	EC	3	4	Absolut

Intern wird der im OPERANDEN adressierte Wert vom X-Register abgezogen, der resultierende Wert jedoch verworfen und nur die Flaggen entsprechend gesetzt. Das X-Register und der Vergleichswert bleiben unverändert. Wenn das X-Register kleiner ist als der Vergleichswert, liegt das Ergebnis unter Null, also im Negativen, wodurch die N-Flagge gesetzt wird (dies gilt nur für vorzeichenbehaftete Zahlen!). Da bei der Subtraktion dann ein Übertrag (Borgen) stattfindet, wird das Carrybit gelöscht.

	X-Reg. < Wert	X-Reg. = Wert	X-Reg. > Wert
N:	(1)	0	(0)
Z:	0	1	0
C:	0	1	1

Üblicherweise steht nach dem Vergleich ein Verzweigungsbefehl. Durch geeignete Wahl (e.v. Kombination) sind alle Bedingungen testbar. Wenn Sie mit BCC oder BCS ein Ergebnis testen wollen, sollten Sie nie mit CPX #$00 vergleichen, da dann keine echte Entscheidung möglich ist (jeder Wert ist >= 00, setzt also das Carrybit). CPX wird häufig in Zählschleifen verwendet, die nicht bei 0 oder 1 enden sollen (können).

Beispiel: Zählschleife bis zu einem def. Wert

```
        LDX #00
SCHLEIFE ....
        ....
        INX
        CPX #WERT
        BLT SCHLEIFE ;noch kleiner (= BCC)
        ....
```

CPY

ComPare with Y-register
Vergleiche mit dem Y-Register

Das Y-Register wird mit dem im OPERANDEN spezifizierten Wert verglichen und die Flaggen N, Z und C entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
CPY #.	C0	2	2	Unmittelbar
CPY .	C4	2	3	Zero-Page
CPY ..	CC	3	4	Absolut

Intern wird der im OPERANDEN adressierte Wert vom Y-Register abgezogen, der resultierende Wert jedoch verworfen und nur die Flaggen entsprechend gesetzt. Das Y-Register und der Vergleichswert bleiben unverändert. Wenn das Y-Register kleiner ist als der Vergleichswert, liegt das Ergebnis unter Null, also im Negativen, wodurch die N-Flagge gesetzt wird (dies gilt nur für vorzeichenbehaftete Zahlen!). Da bei der Subtraktion dann ein Übertrag (Borgen) stattfindet, wird das Carrybit gelöscht.

	Y-Reg. < Wert	Y-Reg. = Wert	Y-Reg. > Wert
N:	(1)	0	(0)
Z:	0	1	0
C:	0	1	1

Üblicherweise steht nach dem Vergleich ein Verzweigungsbefehl. Durch geeignete Wahl (e.v. Kombination) sind alle Bedingungen testbar. Wenn Sie mit BCC oder BCS ein Ergebnis testen wollen, sollten Sie nie mit CPY #$00 vergleichen, da dann keine echte Entscheidung möglich ist (jeder Wert ist >= 00, setzt also das Carrybit). CPY wird häufig in Zählschleifen verwendet, die nicht bei 0 oder 1 enden sollen (können).

Beispiel: Zählschleife bis zu einem def. Wert

```
        LDY #00
SCHLEIFE ....
        ....
        INY
        CPY #WERT
        BLT SCHLEIFE ;noch kleiner (= BCC)
        ....
```

DEA

DEcrement Accumulator
Dekrementiere den Akkumulator um 1

DEA vermindert den Wert des Akkumulators um Eins. Die N- und Z-Flagge werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
DEA	3A	1	2	Implizit (65C02)

Dieser Befehl ist einfacher und schneller als eine Subtraktion (SBC) von 1, aber leider nur im 65C02 vorhanden. Enthält der Akkumulator ursprünglich den Wert $00, ist das Resultat $FF.

DEC

DECrement memory
Dekrementiere den Speicher um 1

DEC vermindert den Wert des im OPERANDEN adressierten Speichers um Eins. Die N- und Z-Flaggen werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
DEC .	C6	2	5	Zero-Page
DEC .,X	D6	2	6	Zero-Page indiziert mit X
DEC ..	CE	3	6	Absolut
DEC ..,X	DE	3	7/6	Absolut indiziert mit X

Dieser Befehl ist einfacher und schneller als eine Subtraktion (SBC) von 1. Enthält der Speicherplatz ursprünglich den Wert $00, ist das Resultat $FF. DEC wird auch häufig benutzt, um Zeiger (Pointer) herunterzuzählen oder um das höherwertige Byte eines 16-Bit Wertes bei der Subtraktion eines 8-Bit Wertes zu korrigieren.

Beispiel: Subtraktion 16-Bit Wert1 - 8-Bit Wert2

```
        SEC
        LDA WERT1lo
        SBC WERT2
        STA WERT1lo
        BCS OK
        DEC WERT1hi
OK ....
```

DEX

DEcrement X-register
Dekrementiere das X-Register um 1

DEX vermindert den Wert des X-Registers um Eins. Die N- und Z-Flagge werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
DEX	CA	1	2	Implizit

Enthält das X-Register ursprünglich den Wert $00, ist das Resultat $FF. DEX wird häufig in Zählschleifen benutzt und um Indexzeiger herunterzuzählen, mit denen Datenblöcke adressiert werden.
Beispiel: Gebe X Leerzeichen aus

```
            LDX ANZAHL
            LDA #$AO
SCHLEIFE    JSR COUT ; $FDED
            DEX
            BNE SCHLEIFE
```

DEY

DEcrement Y-register
Dekrementiere das Y-Register um 1

DEY vermindert den Wert des Y-Registers um Eins. Die N- und Z-Flagge werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
DEY	88	1	2	Implizit

Enthält das Y-Register ursprünglich den Wert $00, ist das Resultat $FF. DEY wird häufig in Zählschleifen benutzt und um Indexzeiger herunterzuzählen, mit denen Datenblöcke adressiert werden.
Beispiel: Löschen von 255 Bytes im Speicher (nicht 256!)

```
            LDY #$FF
            LDA #$00
SCHLEIFE    STA SPEICHER,Y
            DEY
            BNE SCHLEIFE
```

EOR

Exclusive-OR with accumulator
Exklusiv-ODER verknüpfen mit dem Akkumulator

Der im OPERANDEN angegebene Wert wird bitweise mit dem Akkumulator Exklusiv-ODER verknüpft. Die N- und Z-Flaggen werden entsprechend dem Gesamtergebnis gesetzt, das im Akkummulator verbleibt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
EOR #.	49	2	2	Unmittelbar
EOR .	45	2	3	Zero-Page
EOR .,X	55	2	4	Zero-Page indiziert mit X
EOR ..	4D	3	4	Absolut
EOR ..,X	5D	3	4*	Absolut indiziert mit X
EOR ..,Y	59	3	4*	Absolut indiziert mit Y
EOR (.,X)	41	2	6	Indiziert indirekt mit X
EOR (.),Y	51	2	5*	Indirekt indiziert mit Y
EOR (.)	52	2	5	Indirekt (65C02)

EOR bedeutet, daß das Ergebnis nur dann 1 ist, wenn eines der beiden verknüpften Bits 1 ist. Andernfalls ist das Ergebnis 0.
Der EOR-Befehl wird hauptsächlich zum Testen bestimmter Bitmuster und für Maskierungen benutzt.
Wahrheitstabelle:

	0	1
0	0	1
1	1	0

EOR

Abb. 32: EOR

EOR #$FF komplementiert den Inhalt des Akkumulators, wobei aus jeder 0 eine 1 und aus jeder 1 eine 0 wird. Wenden Sie EOR #$FF nocheinmal an, erhalten Sie den ursprünglichen Inhalt zurück. Auf dieser Technik basieren einige einfache Verschlüsselungstechniken.
Sie können mit EOR auch eine Komplementierungsmaske bauen. Die Maske muß eine 1 an allen Positionen enthalten, an denen das Originalbit komplementiert werden soll, und eine Null an den Stellen, die unverändert bleiben sollen (EOR #$00 bewirkt nichts).
Beispiel: Zweierkomplement einer Zahl

```
LDA ZAHL
EOR #$FF
CLC
ADC #$01
STA KOMPLEMENT
```

INA

INcrement Accumulator
Inkrementiere den Akkumulator um 1

INA erhöht den Wert des Akkumulators um Eins. Die N- und Z-Flaggen werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
INA	1A	1	2	Implizit (65C02)

Dieser Befehl ist einfacher und schneller als eine Addition (ADC) von 1, aber leider nur im 65C02 vorhanden. Enthält der Akkumulator ursprünglich den Wert $FF, ist das Resultat $00, ohne daß das Carrybit gesetzt wird.

INC

INCrement memory
Inkrementiere den Speicher um 1

INC erhöht den Wert des im OPERANDEN adressierten Speichers um Eins. Die N- und Z-Flaggen werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
INC .	E6	2	5	Zero-Page
INC .,X	F6	2	6	Zero-Page indiziert mit X
INC ..	EE	3	6	Absolut
INC ..,X	FE	3	7/6	Absolut indiziert mit X

Dieser Befehl ist einfacher und schneller als eine Addition (ADC) von 1. Enthält der Speicherplatz ursprünglich den Wert $FF, ist das Resultat $00. Dabei wird (im Gegensatz zu ADC) das Carrybit **nicht** gesetzt. INC wird auch häufig benutzt, um Zeiger (Pointer) weiterzuzählen oder um das höherwertige Byte eines 16-Bit Wertes bei der Addition eines 8-Bit Wertes zu korrigieren.
Beispiel: Addition 16-Bit Wert1 + 8-Bit Wert2

```
    CLC
    LDA WERT1lo
    ADC WERT2
    STA WERT1lo
    BCC OK
    INC WERT1hi
OK  ....
```

INX

INcrement X-register
Inkrementiere das X-Register um 1

INX erhöht den Wert des X-Registers um Eins. Die N- und Z-Flaggen werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
INX	E8	1	2	Implizit

Enthält das X-Register ursprünglich den Wert $FF, ist das Resultat $00. INX wird häufig in Zählschleifen benutzt und um Indexzeiger heraufzuzählen, mit denen Datenblöcke adressiert werden.
Beispiel: 256-Byte verschieben (kopieren)

```
          LDX #$00
SCHLEIFE  LDA QUELLE,X
          STA ZIEL,X
          INX
          BNE SCHLEIFE
```

INY

INcrement Y-register
Inkrementiere das Y-Register um 1

INY erhöht den Wert des Y-Registers um Eins. Die N- und Z-Flaggen werden entsprechend dem Ergebnis gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
INY	C8	1	2	Implizit

Enthält das Y-Register ursprünglich den Wert $FF, ist das Resultat $00. INY wird häufig in Zählschleifen benutzt und um Indexzeiger heraufzuzählen, mit denen Datenblöcke adressiert werden. Besondere Bedeutung kommt INY im Zusammenhang mit der indirekt indizierten Adressierung anderer Befehle zu.

Beispiel: Kopieren von 8K (HGR1 -> HGR2)

```
            LDY #$00
            STY QUELLE ;lo
            STY ZIEL   ;lo
            LDA #$20   ;HGR1
            STA QUELLE+1 ;hi
            ASL        ;* 2 = HGR2
            STA ZIEL+1 ;hi
SCHLEIFE    LDA (QUELLE),Y
            STA (ZIEL),Y
            INY
            BNE SCHLEIFE
            INC QUELLE+1
            LDX ZIEL+1
            INX
            STX ZIEL+1
            CPX #$60   ;Ende bei $6000
            BLT SCHLEIFE
            RTS
```

JMP

JuMP to adress
Springe zur angegebenen Adresse

JMP bewirkt einen Sprung zu der durch den OPERANDEN angegebenen Adresse. Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
JMP ..	4C	3	3	Absolut
JMP (..,X)	7C	3	6	Indiziert indirekt mit X (65C02)
JMP (..)	6C	3	5/6	Indirekt

JMP wirkt wie ein GOTO in Basic. Die angegebene Adresse wird in den Program-Counter geladen und das Programm dort weiter abgearbeitet. Von der Adressierung her hat JMP einige Besonderheiten: der Befehl ist *immer* 3 Bytes lang, auch wenn die Adresse auf der Zero-Page liegt. Bei den indirekten Sprüngen darf die Basisadresse 2 Bytes lang sein.
Der indirekte Sprung hat beim 6502 (nicht beim 65C02!) einen schwerwiegenden Fehler: Wenn das Lo-Byte das letzte Byte einer Speicherseite ist (Adresse: $xxFF) und somit das Hi-Byte das erste Byte der folgenden Seite ist (Adresse: $xy00), so zählt der Prozessor intern nicht richtig und holt das Hi-Byte vom ersten Byte derselben Seite (Adresse: $xx00). Nur wenn Lo- und Hi-Byte zur selben Seite gehören (was sicherlich meistens der Fall ist), geht alles wie geplant. Die indirekte Adressierung eignet sich für Sprungtabellen oder Programmiertricks mit relocativem Code. Besonders der neue indiziert indirekte Sprung des 65C02 ermöglicht es, verschiedene Sprünge je nach Wert des X-Registers auszuführen.

Beispiel: Nur eine Demo (gibt 3 A auf den Bildschirm)

```
        LDX #<COUT
        LDY #>COUT
        STX ZEIGER
        STY ZEIGER+1
        LDA #"A"
        JSR JUMP1
        JSR JUMP2
        LDX #$00
        JSR JUMP3
        RTS
JUMP1   JMP COUT
JUMP2   JMP (ZEIGER)
JUMP3   JMP (ZEIGER,X)   ;nur 65C02 !
```

JSR

Jump to SubRoutine
Springe zum Unterprogramm

Der Inhalt des Program Counters wird um 2 erhöht und als Rücksprungadresse auf den Stack gebracht (erst Hi-Byte, dann Lo-Byte). (Anmerkung: dies ist die Adresse des folgenden Befehls *minus* 1) Dann wird die angegebene Unterprogrammadresse in den Program Counter geladen und die Befehlsausführung dort fortgesetzt. Flaggen werden nicht beeinflußt. JSR entspricht einem GOSUB in Basic.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
JSR ..	20	3	6	Absolut

Jedes Unterprogramm sollte mit RTS enden, von wenigen „Tricks" abgesehen. Nach dem RTS wird das Hauptprogramm wieder an der alten Stelle aufgenommen. JSR's werden sehr häufig in Assemblerprogrammen benutzt, da diese zweckmäßig in viele Unterprogramme aufgeteilt werden, die eine bestimmmte Aufgabe erledigen und von verschiedenen Stellen des Hauptprogramms aufgerufen werden. Auch wenn in Hochsprachen viele GOTO's und GOSUB's verpönt sind, so kommt die Maschinensprache doch nicht ohne JMP's, JSR's und Branches aus.
(Was aber *nicht* heißt, daß hier Chaos herrscht!)
Auch mit JSR sind einige Tricks möglich, da hier die Möglichkeit besteht, aus dem laufenden Programm heraus die augenblickliche absolute Lage im Speicher zu bestimmmen (siehe TSX).

LDA

LoaD Accumulator
Lade den Akkumulator

Der Inhalt des mit dem OPERANDEN gewählten Speicherbytes wird in den Akkumulator geladen. Die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
LDA #.	A9	2	2	Unmittelbar
LDA .	A5	2	3	Zero-Page
LDA .,X	B5	2	4	Zero-Page indiziert mit X
LDA ..	AD	3	4	Absolut
LDA ..,X	BD	3	4*	Absolut indiziert mit X
LDA ..,Y	B9	3	4*	Absolut indiziert mit Y
LDA (.,X)	A1	2	6	Indiziert indirekt mit X
LDA (.),Y	B1	2	5*	Indirekt indiziert mit Y
LDA (.)	B2	2	5	Indirekt (65C02)

Der Akkumulator ist das Zentrum des 6502. Fast alle Datenströme laufen durch ihn, indem Speicherstellen geladen, die Werte eventuell manipuliert und wieder irgendwo gespeichert werden. Beispiele für LDA finden Sie fast auf jeder Seite dieser Übersicht.

LDX

LoaD X-register
Lade das X-Register

Der Inhalt des mit dem OPERANDEN gewählten Speicherbytes wird in das X-Register geladen. Die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
LDX #.	A2	2	2	Unmittelbar
LDX .	A6	2	3	Zero-Page
LDX .,Y	B6	2	4	Zero-Page indiziert mit Y
LDX ..	AE	3	4	Absolut
LDX ..,Y	BE	3	4*	Absolut indiziert mit Y

Das X-Register wird mit Daten geladen, z.B. um einen Indexwert zu initialisieren. Beispiele treffen Sie überall in diesem Kapitel an.

LDY

LoaD Y-register
Lade das Y-Register

Der Inhalt des mit dem OPERANDEN gewählten Speicherbytes wird in das Y-Register geladen. Die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
LDY #.	A0	2	2	Unmittelbar
LDY .	A4	2	3	Zero-Page
LDY .,X	B4	2	4	Zero-Page indiziert mit X
LDY ..	AC	3	4	Absolut
LDY ..,X	BC	3	4*	Absolut indiziert mit X

Das Y-Register wird mit Daten geladen, z.B. um einen Indexwert zu initialisieren. Beispiele treffen Sie überall in diesem Kapitel an.

LSR

Logical Shift Right
Logische Verschiebung nach rechts

Alle Bits des Akkumulators bzw. des bezeichneten Speicherplatzes werden um eine Stelle nach rechts geschoben. In das frei werdende Bit 7 kommt eine Null, das alte Bit 0 landet im Carrybit (C-Flagge). Die Flaggen N, Z und C werden entsprechend gesetzt.

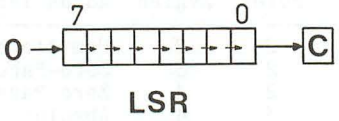


Abb. 33: LSR

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
LSR	4A	1	2	Akkumulator
LSR .	46	2	5	Zero-Page
LSR .,X	56	2	6	Zero-Page indiziert mit X
LSR ..	4E	3	6	Absolut
LSR ..,X	5E	3	7/6	Absolut indiziert mit X

LSR wird hauptsächlich für die Division eines Wertes durch 2 benutzt, da jede Verschiebung nach rechts den Wert halbiert. Zum anderen kann mit LSR leicht geprüft weden, ob eine Zahl gerade oder ungerade ist.
Beispiel: Test gerade/ungerade

```
LDA ZAHL
LSR
BCC GERADE
BCS UNGERADE
```

NOP

No OPeration
Keine Operation

Dieser Befehl tut nichts, außer 1 Byte zu belegen und 2 Prozessortakte zu verbrauchen. Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
NOP	EA	1	2	Implizit

NOP wird hauptsächlich benutzt, um Platz im Code freizuhalten (bei selbstmodifizierenden Programmen), Befehle in bestehenden Programmen zu überschreiben („patchen") oder bei extrem zeitkritischen Routinen (z.B. DOS) einen Feinabgleich vorzunehmen (1 NOP = 2 Mikrosekunden).
Beispiel: Patch für 3 Byte (Überspringen eines JSR's)

```
alt:    .....                neu:    .....
          JSR $ABCD                    NOP
          .....                        NOP
                                       NOP
                                       .....
```

ORA

inclusive OR with Accumulator
Inklusiv-ODER verknüpfen mit dem Akkumulator

Der im OPERANDEN angegebene Wert wird bitweise mit dem Akkumulator Inklusiv-ODER verknüpft. Die N- und Z-Flaggen werden entsprechend dem Gesamtergebnis gesetzt, das im Akkummulator verbleibt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
ORA #.	09	2	2	Unmittelbar
ORA .	05	2	3	Zero-Page
ORA .,X	15	2	4	Zero-Page indiziert mit X
ORA ..	0D	3	4	Absolut
ORA ..,X	1D	3	4*	Absolut indiziert mit X
ORA ..,Y	19	3	4*	Absolut indiziert mit Y
ORA (.,X)	01	2	6	Indiziert indirekt mit X
ORA (.),Y	11	2	5*	Indirekt indiziert mit Y
ORA (.)	12	2	5	Indirekt (65C02)

ORA bedeutet, daß das Ergebnis dann 1 ist, wenn eines oder beide der verknüpften Bits 1 sind. Nur wenn beide Bits 0 sind, ist auch das Ergebnis 0. Der ORA-Befehl wird hauptsächlich zum Testen bestimmter Bitmuster und für Maskierungen benutzt, z.B. um Bit 7 in einem Byte zu setzen. Jede 1 in der Maske setzt das entsprechende Bit im Akkumulator.

Wahrheitstabelle:

	0	1
0	0	1
1	1	1

OR

Abb. 34: ORA

Beispiel: Bit 7 setzen für Ausgabe auf den Bildschirm

```
LDA ASCII
ORA #$80 ; %1000 0000
JSR COUT
....
```

PHA

PusH Accumulator
Schiebe den Akkumulator auf den Stack

Der Inhalt des Akkumulators wird auf den Stack gebracht, der Stackpointer (Stapelzeiger) um eine Position nach unten gesetzt. Es werden keine Flaggen beeinflußt, und auch der Akkumulatorwert bleibt unverändert.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PHA	48	1	3	Implizit

PHA eignet sich mit PLA zusammen dazu, wenige Bytes für kurze Zeit zwischenzuspeichern. Generell muß jedes mit PHA auf den Stack gebrachte Byte von dort mit PLA wieder abgeholt werden. Dies muß spätestens dann geschehen sein, wenn ein RTS folgt, da dieser Befehl zwei Bytes als Rücksprungadresse vom Stack holt. Haben Sie dann dort noch ein Byte „vergessen", landet der Rücksprung an unvorhersehbarer Stelle. Es gibt allerdings einen Trick, mit 2 PHA-Befehlen eine künstliche Rücksprungadresse (ohne vorheriges JSR) auf den Stack zu legen und mit einem RTS (statt eines JMP's) dann dorthin zu springen. Dies ist besonders bei Menüs nützlich, wo der endgültige Sprung von der Wahl des Benutzers abhängt.

Beispiel: Pseudo-Sprung (X enthält die Auswahl 0,2,4,...)

```
        LDA ZIELE+1,X ;Hi-Byte
        PHA
        LDA ZIELE,X   ;Lo-Byte
        PHA
        RTS  ; springt zur angegebenen Adresse + 1
        ......
ZIELE   ADR ZIEL1-1
        ADR ZIEL2-1
        ADR ZIEL3-1
        ......
```

PHP

PusH Processor status
Schiebe den Prozessorstatus auf den Stack

Der Inhalt des Statusregisters wird auf den Stack gebracht, der Stackpointer (Stapelzeiger) um eine Position nach unten gesetzt. Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PHP	08	1	3	Implizit

Da das Statusregister durch PHP in den normalen RAM-Speicher gebracht wird, kann jedes Bit getestet werden, auch die, für die kein direkter Befehl zur Verfügung steht. Zum zweiten wird so der Zustand des Prozessorstatus „gerettet", um ihn einige Befehle später zu testen. Der mit PHP auf den Stack gebrachte Status hat übrigens **immer** Bit 4 (Break-Flagge) gesetzt, so daß ein Test dieses Bits sinnlos ist. Wie auch bei PHA sollten Sie jedes mit PHP auf den Stack gebrachte Byte wieder rechtzeitig entfernen, z.B. mit PLP oder PLA, um nicht „sonderbare Dinge" zu erleben.

Beispiel: Drucken eines Strings mit gesetztem Bit 7 im letzten Byte

```
            LDY #$00
SCHLEIFE    LDA STRING,Y
            PHP  ; Status retten
            ORA #$80 ;Bit 7 setzen
            JSR COUT ;ausgeben
            PLP  ; Status zurückholen
            BMI ENDE ; Stringende erreicht
            INY
            BNE SCHLEIFE
ENDE        RTS
```

PHX

PusH X-register
Schiebe das X-Register auf den Stack

Der Inhalt des X-Registers wird auf den Stack gebracht, der Stackpointer (Stapelzeiger) um eine Position nach unten gesetzt. Es werden keine Flaggen beeinflußt, und auch das X-Register bleibt unverändert.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PHX	DA	1	3	Implizit (65C02)

PHX eignet sich mit PLX zusammen dazu, wenige Bytes für kurze Zeit zwischenzuspeichern. Generell muß jedes mit PHX auf den Stack gebrachte Byte von dort wieder abgeholt werden. Dies muß spätestens dann geschehen sein, wenn ein RTS folgt, da dieser Befehl zwei Bytes als Rücksprungadresse vom Stack holt. Haben Sie dann dort noch ein Byte „vergessen", landet der Rücksprung an unvorhersehbarer Stelle.

PHY

PusH Y-register
Schiebe das Y-Register auf den Stack

Der Inhalt des Y-Registers wird auf den Stack gebracht, der Stackpointer (Stapelzeiger) um eine Position nach unten gesetzt. Es werden keine Flaggen beeinflußt, und auch das Y-Register bleibt unverändert.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PHY	5A	1	3	Implizit (65C02)

PHY eignet sich mit PLY zusammen dazu, wenige Bytes für kurze Zeit zwischenzuspeichern. Generell muß jedes mit PHY auf den Stack gebrachte Byte von dort wieder abgeholt werden. Dies muß spätestens dann geschehen sein, wenn ein RTS folgt, da dieser Befehl zwei Bytes als Rücksprungadresse vom Stack holt. Haben Sie dann dort noch ein Byte „vergessen", landet der Rücksprung an unvorhersehbarer Stelle.

PLA

PuLl Accumulator
Hole den Akkumulatorinhalt vom Stack

PLA holt ein Byte vom Stack in den Akkumulator, der Stackpointer (Stapelzeiger) wird um eine Position nach oben gesetzt. Die N- und Z-Flaggen werden entsprechend dem Wert des geholten Bytes gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PLA	68	1	4	Implizit

PLA ist das Gegenstück zu PHA und kann somit zwischengespeicherte Bytes wieder vom Stack holen. Mit zwei PLA können Sie auch eine Rücksprungadresse vom Stack ziehen, wenn Sie die zuletzt aufgerufene Unterroutine nicht regulär beenden wollen. Dies entspricht dem Applesoftbefehl POP.
Beispiel: POP einer Unterroutine

```
            JSR UNTERPROG.
            RTS  ; normales Ende
UNTERPROG.  .....
            PLA
            PLA
            RTS ; endet jetzt hier
```

PLP

PuLl Processor status
Hole den Prozessorstatus vom Stack

Der Inhalt des Statusregisters wird vom Stack geholt, der Stackpointer (Stapelzeiger) um eine Position nach oben gesetzt. **Alle** Flaggen werden beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PLP	28	1	4	Implizit

PLP ist das Gegenstück zu PHP und dient normalerweise dazu, einen auf den Stack geretteten Status wieder zurückzuladen. Sie können allerdings auch mit PHA erst einen Wert auf den Stack bringen und ihn dann mit PLP in den Status holen, um hier von Ihnen gewünschte Bits künstlich zu setzen oder zu löschen. Denken Sie auch hier daran, daß Sie nur so viele Bytes vom Stack holen, wie Sie vorher darauf getan haben.
Beispiel: Status auf Null setzen

```
            LDA #$00
            PHA
            PLP
```

PLX

PuLl X-register
Hole das X-Register vom Stack

PLX holt ein Byte vom Stack in das X-Register, der Stackpointer (Stapelzeiger) wird um eine Position nach oben gesetzt. Die N- und Z-Flaggen werden entsprechend dem Wert des geholten Bytes gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PLX	FA	1	4	Implizit (65C02)

PLX ist das Gegenstück zu PHX und kann somit zwischengespeicherte Bytes wieder vom Stack holen.

PLY

PuLl Y-register
Hole das Y-Register vom Stack

PLY holt ein Byte vom Stack in das Y-Register, der Stackpointer (Stapelzeiger) wird um eine Position nach oben gesetzt. Die N- und Z-Flaggen werden entsprechend dem Wert des geholten Bytes gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
PLY	7A	1	4	Implizit (65C02)

PLY ist das Gegenstück zu PHY und kann somit zwischengespeicherte Bytes wieder vom Stack holen.
Beispiel: Retten aller Register

```
PHP
PHA
PHX
PHY
TSX
PHX
....
PLX  ;in umgekehrter Reihenfolge
TXS  ;wieder alle Register vom
PLY  ;Stack holen
PLX
PLA
PLP
```

ROL

ROtate Left
Rotiere nach links mit Übertrag

ROL rotiert jedes Bit des Akkumulators oder des gewählten Speicherplatzes um eine Position nach links. Das (alte) Carrybit wandert in das freiwerdende Bit 0, das (alte) Bit 7 wird zum neuen Carrybit. Es bewegen sich also 9 Bit. Das Endergebnis setzt ferner die N- und Z-Flaggen je nach Resultat.

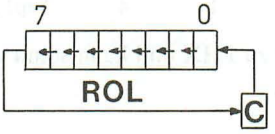


Abb. 35: ROL

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
ROL	2A	1	2	Akkumulator
ROL .	26	2	5	Zero-Page
ROL .,X	36	2	6	Zero-Page indiziert mit X
ROL ..	2E	3	6	Absolut
ROL ..,X	3E	3	7/6	Absolut indiziert mit X

ROL hat seine Hauptanwendung in Multiplikations- und Divisionsroutinen, da der 6502 diese nicht als Befehle eingebaut hat. Zusammen mit ASL ist es auch möglich, einen 16-Bit Wert nach links zu schieben (nicht rotieren!). Dies wird als Doppelschieben (double shift) bezeichnet.

Beispiel: Doppelschieben eines 16-Bit Wertes um eine Position

```
ASL WERTlo
ROL WERThi
```

ROR

ROtate Right
Rotiere nach rechts mit Übertrag

ROR rotiert jedes Bit des Akkumulators oder des gewählten Speicherplatzes um eine Position nach rechts. Das (alte) Carrybit wandert in das freiwerdende Bit 7, das (alte) Bit 0 wird zum neuen Carrybit. Es bewegen sich also 9 Bit. Das Endergebnis setzt ferner die N- und Z-Flaggen je nach Resultat.

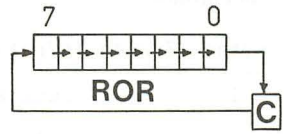


Abb. 36: ROR

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
ROR	6A	1	2	Akkumulator
ROR .	66	2	5	Zero-Page
ROR .,X	76	2	6	Zero-Page indiziert mit X
ROR ..	6E	3	6	Absolut
ROR ..,X	7E	3	7/6	Absolut indiziert mit X

ROR hat seine Hauptanwendung in Multiplikations- und Divisionsroutinen, da der 6502 diese nicht als Befehle eingebaut hat. Zusammen mit LSR ist es auch möglich, einen 16-Bit Wert nach rechts zu schieben (nicht rotieren!). Dies wird als Doppelschieben (double shift) bezeichnet.

Beispiel: Doppelschieben eines 16-Bit Wertes um eine Position

```
LSR WERThi
ROR WERTlo
```

RTI

ReTurn from Interrupt
Rückkehr von der Unterbrechungsroutine

RTI holt ein Byte vom Stack und läd es in den Prozessorstatus. Danach werden zwei weitere Bytes geholt, die in den Program Counter (erst Lo-Byte, dann Hi-Byte) wandern. Der Stackpointer (Stapelzeiger) wird dabei um drei Positionen nach oben geschoben. Die weitere Programmausführung beginnt beim Wert des Program Counters, zu dem vorher **nicht** 1 addiert wurde (siehe auch RTS). Alle Flaggen werden beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
RTI	40	1	6	Implizit

Wenn ein Interrupt beim Apple ausgelöst wird, bringt der Prozessor automatisch eine Rückkehradresse und den Prozessorstatus auf den Stack. RTI ist die Umkehrung dieses Vorgangs, um nach dem Ende der Unterbrechungsroutine wieder an der alten Stelle fortzufahren. RTI ist **nicht** dasselbe wie PLP + RTS, da bei RTI der Program Counter nicht um 1 erhöht wird.

RTS

ReTurn from Subroutine
Rückkehr von der Unterroutine

RTS holt zwei Byte vom Stack in den Program Counter (erst Lo-Byte, dann Hi-Byte) und erhöht diesen Wert dann um Eins. Der Stackpointer (Stapelzeiger) wird um zwei Positionen nach oben geschoben und das Programm an der um 1 erhöhten Adresse fortgesetzt. Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
RTS	60	1	6	Implizit

RTS entspricht dem RETURN bei Basic, da es dazu dient, am Ende eines Unterprogrammes zum Hauptprogramm zurückzukehren. Jedes Unterprogramm muß wenigstens ein RTS enthalten, das die durch JSR auf den Stack gerettete Rücksprungadresse wieder zurückholt.
Mit 2 PHA-Befehlen kann ein künstlicher Rücksprung definiert werden, mit 2 PLA-Befehlen kann ein Rücksprung vom Stack gezogen werden (POP).
Beispiel: siehe PLA und PHA

SBC

SuBtract with Carry
Subtrahiere mit Übertrag vom Akkumulator

SBC subtrahiert den Wert des mit dem OPERANDEN gewählten Speicherbytes vom Inhalt des Akkumulators. *Zusätzlich* wird eine 1 subtrahiert, wenn das Carrybit (C-Flagge) *gelöscht* ist. Das Ergebnis findet sich im Akkumulator, das Carrybit enthält den neuen Übertrag. Die Flaggen N, V, Z und C werden entsprechend gesetzt.
SBC arbeitet sowohl im Dezimal- als auch im Binärmodus. Die Flaggen sind beim 6502 nur im Binärmodus gültig, beim 65C02 in beiden Modi.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
SBC #.	E9	2	2	Unmittelbar
SBC .	E5	2	3	Zero-Page
SBC .,X	F5	2	4	Zero-Page indiziert mit X
SBC ..	ED	3	4	Absolut
SBC ..,X	FD	3	4*	Absolut indiziert mit X
SBC ..,Y	F9	3	4*	Absolut indiziert mit Y
SBC (.,X)	E1	2	6	Indiziert indirekt mit X
SBC (.),Y	F1	2	5*	Indirekt indiziert mit Y
SBC (.)	F2	2	5	Indirekt (65C02)

SBC ist der einzige Subtraktionsbefehl des 6502. Um ohne Übertrag zu subtrahieren, muß vor jeder Rechnung das Carrybit mit SEC gesetzt werden.
Die Funktion des Carrybits ist auf den ersten Blick etwas verwirrend, da es sich scheinbar anders als bei der Addition mit ADC verhält. Vielleicht fällt es Ihnen leichter, sich das Carrybit als „Borge-Flagge" bei der Subtraktion vorzustellen. Diese „Borge-Flagge" wird vor einer Subtraktion mit einer 1 „gefüllt". Wenn die folgende Subtraktion ein Ergebnis liefert, das größer oder gleich Null ist, ist kein Borgen erforderlich, die „Borge-Flagge" bleibt 1 und es ist keine Ergebniskorrektur notwendig (kein Übertrag). Liegt das Ergebnis dagegen unter Null, wird die 1 aus der „Borge-Flagge" geborgt und damit auf 0 gesetzt. Bei der nachfolgenden Subtraktion ist wegen des nun bestehenden Übertrags eine Korrektur erforderlich, zusätzlich wird eine 1 abgezogen. Die 0 in der „Borge-Flagge" zeigt damit, daß eine Eins mehr subtrahiert werden muß, eine 1 in der „Borge-Flagge" zeigt, daß nichts weiter (0) subtrahiert werden muß. Wenn Sie jetzt statt „Borge-Flagge" wieder Carrybit sagen, sind die Verhältnisse so wie oben beschrieben.

Beispiel: WERT1 - WERT2 = WERT3

```
SEC
LDA WERT1lo
SBC WERT2lo
STA WERT3lo
LDA WERT1hi
SBC WERT2hi
STA WERT3hi
```

SEC

SEt Carry
Setze das Carrybit

SEC setzt das Carrybit im Prozessorstatus auf 1.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
SEC	38	1	2	Implizit

SEC muß vor einer Subtraktion ohne Übertrag (s.o.) ausgeführt werden. Für relocativen Code kann zusammen mit BCS eine erzwungene Verzweigung erreicht werden. Gelegentlich wird das Carrybit auch in Unterroutinen bewußt gesetzt, um nach dem Rücksprung mit RTS dem Hauptprogramm eine „Nachricht" zu übermitteln, z.B. daß ein Fehler aufgetreten ist. Besonders DOS und ProDOS benutzen diese Möglichkeit häufig.
Beispiel: Subtraktion zweier 8-Bit Werte

```
SEC
LDA WERT1
SBC WERT2
BCC FEHLER ;Ergebnis unter Null
....
```

SED

SEt Decimal Mode
Dezimal-Betriebsart einschalten

SED setzt die D-Flagge und schaltet den Prozessor vom Binärmodus in den Dezimalmodus um.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
SED	F8	1	2	Implizit

Die Befehle ADC und SBC werden durch die Betriebsart wesentlich beeinflußt, da sie entweder binäre oder BCD-Ergebnisse liefern. Der „übliche" Modus ist der Binärmodus. Beim 6502 ist der Modus nach dem Einschalten undefiniert. Beim 65C02 wird beim Einschalten und bei jedem RESET auf den Binärmodus geschaltet. Die ROM-Routinen des Apple benutzen den Binärmodus. Der Dezimalmodus wird beim Apple im allgemeinen selten benutzt.

SEI

SEt Interrupt mask
Setze die Unterbrechungssperre

SEI setzt die I-Flagge im Prozessorstatus und blockiert damit die Unterbrechungsmöglichkeit des 6502 (Hardware Interrupt).

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
SEI	78	1	2	Implizit

Eine Interrupt-Anforderung (Interrupt ReQuest IRQ) kann mit SEI abgeblockt werden. Dies geschieht automatisch durch das ROM des Apple, wenn gerade ein IRQ eingetroffen ist. Ein Benutzer kann mit SEI auch zeitweise IRQ's sperren, wenn z.B. zeitkritische Routinen ausgeführt werden müssen, die keine Unterbrechung erlauben. So ist beispielsweise in Teilen des DOS (RWTS) und ProDOS (RWTB) die I-Flagge gesetzt. Mit CLI wird die Sperre wieder aufgehoben. NMI's (Non-Maskable Interrupt) und RESET können mit SEI **nicht** gesperrt werden.

STA

STore Accumulator
Speichere den Akkumulator

Der Inhalt des Akkumulators wird an dem durch den OPERANDEN bezeichneten Speicherplatz abgelegt, der Akkumulator bleibt unverändert. Die Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
STA .	85	2	3	Zero-Page
STA .,X	95	2	4	Zero-Page indiziert mit X
STA ..	8D	3	4	Absolut
STA ..,X	9D	3	5	Absolut indiziert mit X
STA ..,Y	99	3	5	Absolut indiziert mit Y
STA (.,X)	81	2	6	Indiziert indirekt mit X
STA (.),Y	91	2	6	Indirekt indiziert mit Y
STA (.)	92	2	5	Indirekt (65C02)

Der Akkumulator ist das Zentrum des 6502. Fast alle Datenströme laufen durch ihn, indem Speicherstellen geladen, die Werte eventuell manipuliert und wieder irgendwo gespeichert werden. Beispiele für STA finden Sie fast auf jeder Seite dieser Übersicht.

STX

STore X-register
Speichere das X-Register

Der Inhalt des X-Registers wird an dem durch den OPERANDEN bezeichneten Speicherplatz abgelegt, das X-Register bleibt unverändert. Die Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
STX .	86	2	3	Zero-Page
STX .,Y	96	2	4	Zero-Page indiziert mit Y
STX ..	8E	3	4	Absolut

Das X-Register wird des öfteren als Laufvariable benutzt, deren Wert mit STX zwischengespeichert werden kann. Mit LDX/STX können auch Daten verschoben werden, wenn beispielsweise der Akkumulator nicht verändert werden darf.

STY

STore Y-register
Speichere das Y-Register

Der Inhalt des Y-Registers wird an dem durch den OPERANDEN bezeichneten Speicherplatz abgelegt, das Y-Register bleibt unverändert. Die Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
STY .	84	2	3	Zero-Page
STY .,X	94	2	4	Zero-Page indiziert mit X
STY ..	8C	3	4	Absolut

Das Y-Register wird des öfteren als Laufvariable benutzt, deren Wert mit STY zwischengespeichert werden kann. Mit LDY/STY können auch Daten verschoben werden, wenn beispielsweise der Akkumulator nicht verändert werden darf.

Beispiel: Primitiv-Division WERT1/WERT2 = WERT3

```
            CLD
            LDY #$00
SCHLEIFE    INY
            SEC
            LDA WERT1
            SBC WERT2
            STA WERT1
            BEQ ENDE
            BCS SCHLEIFE
            DEY
ENDE        STY WERT3
            RTS
```

STZ

STore Zero into memory
Setze eine Speicherstelle auf Null

Die im OPERANDEN bezeichnete Speicherstelle wird auf Null gesetzt. Die Register A, X und Y bleiben unverändert, Flaggen werden nicht beeinflußt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
STZ .	64	2	3	Zero-Page (65C02)
STZ .,X	74	2	4	Zero-Page indiziert mit X (65C02)
STZ ..	9C	3	4	Absolut (65C02)
STZ ..,X	9E	3	5	Absolut indiziert mit X (65C02)

Dieser neue Befehl des 65C02 ist eine bequeme Möglichkeit, Speicherstellen zu initialisieren (auf Null zu setzen), ohne daß ein Register benutzt werden muß, das dadurch natürlich verändert würde.

Beispiel: Vergleich 6502/65C02

```
alt:    LDA #$00        neu  STZ SPEICHER
        STA SPEICHER
```

TAX

Transfer Accumulator to X-register
Transferiere den Akkumulator in das X-Register

Der Inhalt des Akkumulators wird in das X-Register übertragen. Der Akkumulator bleibt unverändert, die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TAX	AA	1	2	Implizit

Mit TAX kann der Akkumulator in das X-Register übertragen werden, wenn der Inhalt z.B. nicht direkt in das X-Register geladen werden kann oder wenn vorher die Daten noch mit Addition oder Subtraktion manipuliert werden sollen, was ja im X-Register nicht möglich ist. Zusammen mit TYA kann das Y-Register in das X-Register gebracht werden.
Beispiel: Tausch Y -> X

```
TYA ;Y in den Akku
TAX ;Akku nach X
```

Tastaturabfrage für Menüs (vereinfacht)

```
JSR RDKEY ;Tastatur lesen
AND #$0F ;ASCII -› ZAHL
ASL ;* 2
TAX
JMP (ZIEL,X) ;nur 65C02 (!)
```

TAY

Transfer Accumulator to Y-register
Transferiere den Akkumulator in das Y-Register

Der Inhalt des Akkumulators wird in das Y-Register übertragen. Der Akkumulator bleibt unverändert, die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TAY	A8	1	2	Implizit

Mit TAY kann der Akkumulator in das Y-Register übertragen werden, wenn der Inhalt z.B. nicht direkt in das Y-Register geladen werden kann oder wenn vorher die Daten noch mit Addition oder Subtraktion manipuliert werden sollen, was ja im Y-Register nicht möglich ist. Zusammen mit TXA kann das X-Register in das Y-Register gebracht werden.
Beispiel: Tausch X -> Y

```
TXA ;X in den Akku
TAY ;Akku in das Y-Register
```

TRB

Test and Reset Bits
Teste Bits und lösche sie

Die im OPERANDEN bezeichnete Speicherstelle wird mit dem Akkumulator bitweise AND verknüpft und anschließend invertiert. Der Akkumulator bleibt unverändert, die Z-Flagge wird entsprechend dem Ergebnis gesetzt, die N- und V-Flaggen erhalten Bit 7 und 6 des adressierten Bytes (wie bei BIT!).

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TRB .	14	2	5	Zero-Page (65C02)
TRB ..	1C	3	6	Absolut (65C02)

Wenn Sie Schwierigkeiten haben, obige Beschreibung nachzuvollziehen: man kann es im Resultat auch einfacher sagen.
Im adressierten Byte werden die Bits gelöscht, die im Akkumulator gesetzt sind. Nullen im Akkumulator haben keinen Einfluß.
So können Sie z.B. Bit 7 direkt im Speicher löschen, ohne das Byte erst in den Akkumulator holen zu müssen.
Beispiel: Bit 7 löschen in Speicher

```
alt:  LDA SPEICHER          neu: LDA #$80 ;%1000 0000
      AND #$7F ;%0111 1111       TRB SPEICHER
      STA SPEICHER
```

Wenn dieser Vorgang in einer Schleife stattfindet, sparen sie viel Code und Zeit.

TSB

Test and Set Bits
Teste Bits und setze sie

Die im OPERANDEN bezeichnete Speicherstelle wird mit dem Akkumulator bitweise inklusiv-ODER verknüpft. Der Akkumulator bleibt unverändert, die Z-Flagge wird entsprechend dem Ergebnis gesetzt, die N- und V-Flaggen erhalten Bit 7 und 6 des adressierten Bytes (wie bei BIT!).

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TSB .	04	2	5	Zero-Page (65C02)
TSB ..	0C	3	6	Absolut (65C02)

Wenn Sie Schwierigkeiten haben, obige Beschreibung nachzuvollziehen: man kann es im Resultat auch einfacher sagen.
Im adressierten Byte werden auch die Bits gesetzt, die im Akkumulator zusätzlich gesetzt sind. Nullen im Akkumulator haben keinen Einfluß.
So können Sie z.B. Bit 7 direkt im Speicher setzen, ohne das Byte erst in den Akkumulator holen zu müssen.
Beispiel: Bit 7 setzen in Speicher

```
alt:  LDA SPEICHER          neu: LDA #$80 ;%1000 0000
      ORA #$80 ;%1000 0000       TSB SPEICHER
      STA SPEICHER
```

Wenn dieser Vorgang in einer Schleife stattfindet, sparen sie viel Code und Zeit.

TSX

Transfer Stack to X-register
Transferiere den Stackpointer in das X-Register

Der Inhalt des Stackpointers (Stapelzeigers) wird in das X-Register übertragen. Der Stackpointer bleibt unverändert, die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TSX	BA	1	2	Implizit

Dies ist die einzige Operation, die in einem Programm die augenblickliche Lage des Stackpointers zugänglich macht. Sie können diesen Wert zwischenspeichern und später wieder laden (mit TXS), wenn z.B. ein Unterprogramm eine variable Anzahl von Bytes auf dem Stack „vergißt". Ebenso können Sie die Lage des Zeigers verändern, ohne daß Sie etwas auf den Stack schieben oder von dort holen müßten. Da dies sehr riskant ist, sollten Sie solche Manipulationen besser unterlassen.

Beispiel: Stackpointer um ein Byte versetzen

```
TSX
DEX
TXS
```

Obiges Beispiel wird in manchen Routinen benutzt, die selbst ihre absolute Lage im Speicher feststellen müssen (z.B. um festzustellen, in welchem Slot die Karte steckt). Dazu wird eine feste RTS-Instruktion im ROM mit JSR angesprungen und *nach* der Rückkehr die noch im Stackspeicher befindliche Rücksprungadresse gelesen. Falls zwischenzeitlich ein Interrupt passiert, stellen sich allerdings unvorhersehbare Ergebnisse ein.

```
JSR RETURN  ;z.B. $FF58
TSX
LDA STACK,X ;Hi-Byte
STA ADRESSEhi
DEX
LDA STACK,X ;Lo-Byte
STA ADRESSElo
```

ADRESSE (hilo) zeigt auf das dritte Byte des JSR-Befehls!

TXA

Transfer X-register to Accumulator
Transferiere das X-Register in den Akkumulator

Der Inhalt des X-Registers wird in den Akkumulator übertragen. Das X-Register bleibt unverändert, die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TXA	8A	1	2	Implizit

Oft ist es notwendig, den Inhalt des X-Registers weiter zu verarbeiten. Da außer INX und DEX wenig Möglichkeiten im X-Register bestehen, wird das X-Register in den Akkumulator übertragen, dessen vielfältige Operationen dann darauf angewendet werden können. Mit TAX kann schließlich der neue Wert wieder zurück ins X-Register gelangen. Über TXA/TAY kann das X-Register über den Akkumualtor in das Y-Register transferiert werden.

TXS

Transfer X-register to Stack
Transferiere das X-Register in den Stackpointer

Der Inhalt des X-Registers wird in den Stackpointer (Stapelzeiger) übertragen. Das X-Register wird ebenso wenig verändert wie die Flaggen.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TXS	9A	1	2	Implizit

Dies ist die einzige Operation, die in einem Programm willkürlich den Stackpointer versetzt. Wie bei allen Stackbefehlen sollten Sie nur sehr bedacht damit umgehen. Zusammen mit einem vorher benutzten TSX kann der alte Zeigerwert wieder hergestellt werden. Eine wichtige Anwendung ist die Initialisierung des Stacks nach einem schweren Programm- oder Bedienungsfehler, der zum Abbruch führt. Da dann der Stack zumeist noch mit vielen (ungültigen) Daten gefüllt ist, werden diese durch Hochsetzen des Stackpointers „gelöscht".

Beispiel: Stack-Initialisierung nach Fehler

```
(Fehlermeldung ausgeben)
LDX #$F8
TXS  ;Stack hochgesetzt
JMP WARMSTART
```

Diese Routine darf **nicht** mit RTS enden, da sich keine gültige Rücksprungadresse auf dem Stack befindet!

TYA

Transfer Y-register to Accumulator
Transferiere das Y-Register in den Akkumulator

Der Inhalt des Y-Registers wird in den Akkumulator übertragen. Das Y-Register bleibt unverändert, die N- und Z-Flaggen werden entsprechend gesetzt.

MNEMONIC	Opcode	Bytes	Zyklen	Adressierungsart
TYA	98	1	2	Implizit

Oft ist es notwendig, den Inhalt des Y-Registers weiter zu verarbeiten. Da außer INY und DEY wenige Möglichkeiten im Y-Register bestehen, wird das Y-Register in den Akkumulator übertragen, dessen vielfältige Operationen dann darauf angewendet werden können. Mit TAY kann schließlich der neue Wert wieder zurück ins Y-Register gelangen. Mit TYA/TAX kann das Y-Register über den Akkumualtor in das X-Register transferiert werden.

5. ASSESSOR

Der Editor und Assembler

Einführung

ASSESSOR ist ein interaktiver Editor und 6502-Assembler für den Apple II, der auch den neuen 65C02-Prozessor des IIe und IIc unterstützt. ASSESSOR wurde geschaffen, um Ihnen den Einstieg in die Welt der Maschinensprache so leicht wie möglich zu machen. Darüber hinaus wird Ihnen ASSESSOR auch für Ihre weiterführende Assemblerprogrammierung von großem Nutzen sein, denn ASSESSOR weist viele Merkmale von teuren, professionellen Assemblern auf.
ASSESSOR besitzt einen vollwertigen Zeileneditor für die Erstellung des Quellcodes, der schon bei der Eingabe viele Schreibfehler entdeckt und Ihnen so Zeit und Mühe erspart. Umfangreiche Editiermöglichkeiten helfen Ihnen bei der schnellen Überarbeitung. Der Assembler übersetzt sowohl alle 56 Befehle des 6502-Microprozessors als auch die erweiterten Befehle und Adressierungsarten des 65C02. Darüber hinaus versteht ASSESSOR 15 weitere Steuerzeichen (Pseudo-Opcodes), die Ihnen die Arbeit mit Zeichenketten (Strings), Sprungtabellen usw. zu einem Kinderspiel werden lassen. Um möglichst viel Speicherplatz für Ihren Quellcode und den entstehenden Objektcode zu lassen, wurde ASSESSOR als eine kompakte Einheit 100% in Maschinensprache erstellt. Dies kommt zusätzlich der Übersetzungsgeschwindigkeit zugute.
ASSESSOR ist ein speicherorientierter Assembler, der gleichzeitig sowohl den Quell- als auch den Objektcode im RAM hält. Zeitraubende Diskettenzugriffe entfallen damit. Die Aufteilung des Apple-Speichers ist von Ihnen wählbar. Voreingestellt sind ca. 27K für Quellcode und Label-Tabelle sowie ca. 4K für den Objektcode. Das ist ausreichend für Programme mit weit mehr als 1000 Zeilen Quellcode!

ASSESSOR ist ein sogenannter „2-Pass-Assembler", der im ersten Durchgang eine LABEL-Tabelle mit allen nötigen Adressen anlegt und im 2. Durchgang den endgültigen Maschinencode erzeugt. Somit sind Vorwärts- und Rückwärtssprünge möglich, ohne daß Sie irgendwelche Adressen oder Sprungweiten berechnen müssen.
ASSESSOR bietet Ihnen zusätzlich bequeme Möglichkeiten zum Laden und Speichern Ihrer Files. Da auch normale DOS-Textfiles geschrieben und gelesen werden können, ist ein Code-Austausch mit anderen Assemblern wie Merlin, Big Mac oder Lisa ohne große Probleme realisierbar.
Um Ihnen die Bedienung von ASSESSOR so leicht wie möglich zu machen, wurde sie an das Ihnen gewohnte Applesoft angepaßt. Die Befehlsfolgen sind weitgehend identisch mit einem sehr bekannten Applesoft-Editor, auch wenn die interne Realisierung auf völlig unterschiedlichem Wege gelang. Die beste Weise, sich mit ASSESSOR vertraut zu machen, ist die Arbeit mit ASSESSOR. Probieren Sie die einzelnen Beispiele dieses Buches aus, und Sie werden sehen, wie leicht Ihnen mit ASSESSOR die Programmierung fällt.

Systemvoraussetzungen

ASSESSOR benötigt 48K RAM sowie Applesoft im ROM und einen Autostart-Monitor. Damit ist ASSESSOR auf praktisch allen Apple IIplus, IIe und IIc lauffähig. Eine Language-Card oder erweiterte 80-Zeichenkarte wird nicht genutzt, auch wenn sie vorhanden ist.
Um ASSESSOR zu laden, benötigen Sie ein Diskettenlaufwerk mit DOS 3.3 als Betriebssystem. ASSESSOR modifiziert das DOS im Speicher zeitweise für seine eigenen Diskettenzugriffe, wenn etwa Textfiles gelesen werden sollen. Das kann zu Problemen mit einigen DOS-Varianten führen, wenn diese ihrerseits das DOS stark verändern. ASSESSOR wurde erfolgreich getestet mit Apple DOS 3.3, Diversi-DOS 2C und 4C, David-DOS, Pronto-DOS und ZDOS 3.4. Unter ProDOS ist ASSESSOR grundsätzlich *nicht* lauffähig.
Der Editor arbeitet nur mit dem normalen 40-Zeichen Textbildschirm. 80-Zeichenkarten werden nicht unterstützt. Sie können jedoch alle Listings (Quellcode und Quellcode plus Maschinencode) formatiert auf einem Drucker mit 80 Zeichen pro Zeile ausgeben.
Alle Kommandos werden in Großbuchstaben erwartet. Schalten Sie deshalb bei einem IIe oder IIc die Feststelltaste ein. Nur in den Kommentarzeilen des Quellcodes können (und sollten) Sie auch Kleinschrift verwenden.

Der Aufbau von ASSESSOR

ASSESSOR besteht grundsätzlich aus 3 Teilen, die sich alle gleichzeitig im Speicher befinden. Ein Nachladen ist nie notwendig. Der **Kommandointerpreter** versucht, Ihre Tastatureingaben zu entschlüsseln und die gewünschten Funktionen aufzurufen. Versteht er Ihren Befehl nicht (weil Sie sich vielleicht verschrieben haben), erhalten Sie die Meldung „? SYNTAX ERROR". Der **Editor** dient zum Erzeugen und Verändern des Quellcodes. Sie werden die meiste Zeit mit ihm arbeiten. Schließlich übersetzt der **Assembler** den von Ihnen mit dem Editor eingegebenen Quellcode in den ablauffähigen Maschinencode. Einen genauen Speicherbelegungsplan finden Sie auf Seite 209 in diesem Kapitel.

Der Start von ASSESSOR

Für den täglichen Gebrauch sollten Sie eine eigene Diskette für ASSESSOR anlegen, da die Begleitdiskette zu voll ist, um auch noch die von Ihnen geschriebenen Programme zu fassen. Laden Sie den File „START ASSESSOR" mit „LOAD START ASSESSOR" in den Speicher und initialisieren Sie dann mit „INIT START ASSESSOR" eine **neue** Diskette. Nun müssen Sie noch den File „ASSESSOR" auf diese Diskette übertragen. Benutzen Sie dazu ein „File-Transfer-Programm" wie z.B. „FID" von der Apple System Master Diskette oder eines der vielen käuflichen Kopierprogramme.
Legen Sie die so entstandene Diskette in das Laufwerk und geben Sie danach den Befehl „RUN START ASSESSOR". Sie können aber auch neu „booten". Damit rufen Sie automatisch ein kurzes Applesoftprogramm auf, das „MAXFILES 1" für DOS setzt und anschließend das eigentliche Programm unmittelbar unter den einen verbleibenden DOS-Puffer läd. Das Applesoftprogramm wird danach automatisch gelöscht, da es nicht mehr benötigt wird. Starten Sie ASSESSOR **niemals** direkt mit „BRUN ASSESSOR", da dann Teile von ASSESSOR durch das DOS zerstört werden und eine Assemblierung nicht mehr möglich ist, auch wenn der Editorteil noch funktioniert. Nach dem erfolgreichen Start meldet sich ASSESSOR mit einem kurzen Copyright-Hinweis und dem blinkenden Cursor:
ASSESSOR wartet auf Ihre Befehle.

Das Format des Quellcodes

Jede Zeile eines Assemblerprogramms besteht aus vier Feldern, die aber nicht immer alle besetzt sein müssen. Es gibt ein LABEL-Feld, ein MNEMONIC-Feld, ein OPERANDEN-Feld und schließlich ein KOMMENTAR-Feld, die folgendermaßen angeordnet sind:

```
LABEL MNEMONIC OPERAND ;KOMMENTAR
```

Die Felder werden durch wenigstens ein Leerzeichen voneinander getrennt und dadurch von ASSESSOR erkannt. Leerzeichen innerhalb eines Feldes sind daher nicht zulässig, mit zwei Ausnahmem: Bei ASCII-Strings dürfen im OPERANDEN und bei KOMMENTARen überall auch Leerzeichen stehen. Wenn in der ersten Zeilenposition ein Großbuchstabe steht, interpretiert ASSESSOR alle folgenden Zeichen bis zum ersten Leerzeichen als LABEL. Ist diese Position leer, so signalisiert dies ASSESSOR , daß kein LABEL in dieser Zeile vorhanden ist. Steht an der ersten Zeilenposition ein „;“ oder „*“, so wird die gesamte Zeile als KOMMENTAR aufgefaßt. Abgesehen von diesen reinen Kommentarzeilen muß jede Zeile wenigstens ein MNEMONIC enthalten. Die Besetzung der anderen Felder hängt ganz vom Programm ab. Dabei gilt folgende Formatregel: Ein LABEL **muß**, ein KOMMENTAR **kann** und MNEMONIC sowie OPERAND **dürfen nicht** in der ersten Position beginnen. Die maximale Länge einer Zeile umfaßt ca. 240 Zeichen. Sie sollten jedoch nicht mehr als gut 60 Zeichen pro Zeile verwenden, damit bei der Assemblierung noch die volle Zeile incl. Maschinencode in einer Druckzeile Platz hat. Verteilen Sie Kommentare gegebenenfalls auf mehrere Zeilen.

Das LABEL-Feld

Der Editor von ASSESSOR zeigt zwar Zeilennummern an, diese dienen jedoch ausschließlich zu Ihrer Information und als Editierhilfe, wenn Sie Zeilen löschen oder einfügen wollen. Der Assembler ignoriert diese Nummern. Er arbeitet stattdessen mit namentlichen Bezeichnungen, den LABELs oder Marken. Jedes LABEL besteht aus 1 bis 8 Zeichen und beginnt immer in der ersten Zeilenposition. Das erste Zeichen **muß** ein Großbuchstabe sein, die restlichen Zeichen dürfen Buchstaben oder Zahlen enthalten. Sonderzeichen wie „#$%&<>?)...“ sind **nicht** erlaubt, auch sollten Sie deutsche Umlaute vermeiden. Ein LABEL darf auch reservierte Applesoft-Befehle enthalten oder buchstabengleich mit Mnemonics sein. „TEXT“ oder „START“ sind also legale

LABEL. Jedes LABEL darf nur einmal definiert sein, sonst bekommen Sie vom Assembler im 1. Durchlauf die Fehlermeldung „DUPLICATE LABEL".

Das MNEMONIC-Feld

Dieses Feld, das auf beiden Seiten von einem Leerzeichen eingeschlosen sein muß, enthält die drei Buchstaben eines gültigen MNEMONICs. Dazu gehören die 56 gültigen Befehle des 6502-Prozessors, die 10 Erweiterungen des 65C02 sowie die folgenden 15 Pseudo-Opcodes:

EQU, ORG, ASC, INV, FLS, DCI, HEX, DFB, DFS, ADR, END, LST, NLS, BLT,BGE.

Die Eingabe muß in Großbuchstaben erfolgen. Da ASSESSOR alle MNEMONICs intern in 1 Byte langen Token abspeichert, werden Schreibfehler sofort beim Abschließen der Zeile durch die RETURN-Taste mit einem „? SYNTAX ERROR" honoriert.

Das OPERANDEN-Feld

Auch dieses Feld ist beidseitig von Leerzeichen eingeschlossen. Es enthält eine Adresse oder einen Ausdruck sowie alle Informationen über die Adressierungsart. Den erlaubten Ausdrücken und Adressen ist ein eigener Abschnitt gewidmet.

Das KOMMENTAR-Feld

Das KOMMENTAR-Feld muß mit einem „;" beginnen und sollte vom OPERANDEN durch ein Leerzeichen getrennt sein. Wenn das MNEMONIC keinen OPERANDEN besitzt, kann der KOMMENTAR auch dem MNEMONIC folgen. Kommentare dienen lediglich als Gedankenstütze für Sie oder andere, die einmal Ihr Programm verstehen sollen. Machen Sie reichlich von knappen Kommentaren Gebrauch, sonst verstehen Sie nach einem Jahr Ihr eigenes Programm nicht mehr. Das KOMMENTAR-Feld kann, wie schon erwähnt, auch eine ganze Zeile einnehmen, wenn diese in der ersten Spalte mit einem „;" oder „*" beginnt. Das Sternchen wurde nur zugelassen, um kompatibel mit anderen Assemblern zu bleiben. Intern wandelt ASSESSOR alle „*" der 1. Spalte in „;" um, so daß Sie beim Listen der Zeile immmer ein „;" erhalten.

Ausdrücke und Adressierungsarten

Dieser Abschnitt ist kein Kurzlehrgang in Assembler. Welcher Befehl in welchen Adressierungsarten benutzt werden darf, entnehmen Sie bitte den anderen Kapiteln dieses Buches. Hier soll lediglich die legale Form erläutert werden.
Das OPERANDEN-Feld enthält gleich mehrere Informationen. Es teilt dem Assembler mit, welche Adressierungsart gewünscht ist und wo die Werte gefunden werden können (Adresse).
Eine Adresse ist ein ganzzahliger Ausdruck, der nicht größer als 65535 ($FFFF) sein darf. Fließkommazahlen und ähnliches sind nicht zulässig. Der Ausdruck darf aus folgendem bestehen:
1) einem LABEL
2) einer Dezimalzahl
3) einer Hexadezimalzahl (mit davorstehendem $)
4) einer Binärzahl (mit davorstehendem %)
5) dem Zeichen „*", für das die augenblickliche Adresse eingesetzt wird
6) unter bestimmten Bedingungen aus einer Folge von ASCII-Zeichen, die in Hochkommata ' oder in Anführungszeichen " eingeschlossen sein müssen
7) den arithmetischen Zeichen „+" und „-"
8) den Sonderzeichen „<" und „>".

Wenn ein Ausdruck arithmetische Zeichen (Addition oder Subtraktion) enthält, müssen auch alle Zwischenergebnisse im Bereich von 0 bis 65535 liegen, um die Meldung „? ILLEGAL QUANTITY ERROR" zu vermeiden. „500+400-800" ist erlaubt, „500-800+400" dagegen nicht. Die Abarbeitung erfolgt von links nach rechts, wie Sie es gewohnt sind.
Die Sonderzeichen „<" und „>" haben eine besondere Bedeutung: von dem gesamten Ausdruck, der rechts von ihnen steht, wird nur das niederwertige Byte (bei „<") oder das höherwertige Byte (bei „>") verwendet. Ihre Wirkung ist also zusätzlich die einer Klammer.
>$300+$1FF+$80 ergibt $05, während $300+>$1FF+$80 zu $302 führt.
Diese Sonderzeichen werden speziell bei der Unmittelbaren Adressierung benötigt. Experimentieren Sie ein wenig mit der „Fragezeichen-Funktion", die Ihnen den direkten Zugriff auf die Rechenkapazitäten von ASSESSOR ermöglicht und Ihnen die Wirkung schneller zeigt als 100 weitere Zeilen.
Beispiele für legale Ausdrücke:

```
ENDE-START+$50
$50+%100110
2550-$30
LABEL1+LABEL2-LABEL3
%1010101010-$7F
<ADRESSE-1
>ADRESSE+LABEL
>ADRESSE+%10010101
*+$20
 usw.
```

ASCII-Zeichen sind zulässig nach den MNEMONICs ASC, FLS, INV und DCI. Sie müssen immer von Hochkommata oder Anführungszeichen eingeschlossen sein. Nach ASC und DCI setzt der Assembler für sie nur den ASCII-Wert ein, wobei die Anführungszeichen " ein gesetztes Bit 7 bewirken und das Hochkomma ' ein gelöschtes Bit 7. Nach DCI wird im letzten Zeichen einer ASCII-Folge zusätzlich das Bit 7 invertiert. Nach FLS und INV werden die Codewerte eingesetzt, die beim Apple blinkende bzw. inverse Zeichendarstellung bewirken.

Eine besondere Regelung gilt bei der Unmittelbaren Adressierung (Immediate Mode), die durch ein vorstehendes Nummernkreuz „#" im OPERANDEN gekennzeichnet ist. Hier darf der berechnete Ausdruck den Wert von 255 ($FF) nicht übersteigen. Durch
#<Ausdruck erhalten Sie nur das niederwertige Byte, durch
#>Ausdruck nur das höherwertige Byte des berechneten Ausdrucks.
Außerdem können Sie ein ASCII-Zeichen angeben, daß in Hochkommata bzw. Anführungszeichen direkt nach dem Nummernkreuz stehen muß. Auch hier gilt: Anführungszeichen setzt Bit 7, Hochkomma löscht Bit 7. Nach dem ASCII-Zeichen darf noch ein arithmetischer Ausdruck stehen, solange das Endergebnis kleiner als 255 bleibt.

```
#"A" liefert $C1
#'A' liefert $41
#'A'+$80 liefert $C1
#$80+'A' ist nicht erlaubt und führt bei der Assemblierung zur Fehlermeldung
„MISSING LABEL".
```

Der 6502-Prozessor besitzt 12 Adressierungsarten, die bei ASSESSOR folgender Syntax gehorchen:

Implizit	MNENONIC
Akkumulator	MNEMONIC
Unmittelbar	MNEMONIC #Ausdruck
Zero-Page	MNEMONIC Ausdruck (<256)
indiziert X	MNEMONIC Ausdruck,X
indiziert Y	MNEMONIC Ausdruck,Y
Absolut	MNEMONIC Ausdruck (>255)
indiziert X	MNEMONIC Ausdruck,X
indiziert Y	MNEMONIC Ausdruck,Y
Indirekt	MNEMONIC (Ausdruck)
indiziert X	MNEMONIC (Ausdruck,X)
indiziert Y	MNEMONIC (Ausdruck),Y

Bei der Adressierungsart „Akkumulator" darf im OPERANDEN kein „A" stehen, wie es bei manchen älteren Assemblern der Fall ist. Die Befehle „INcrement Accumulator" und „DEcrement Accumulator" des 65C02-Prozessors müssen als „INA" bzw. „DEA" eingegeben werden. „INC A" und „DEC A" sind **nicht** zulässig. Die Syntax für „Zero-Page" und „Absolute" ist gleich, die Adressierungsart wird vom Assembler anhand der Größe des Ausdrucks bestimmt. Ist der Ausdruck kleiner als 256 ($100), wird immer die Zero-Page Adressierung gewählt, ansonsten die Absolute Adressierung. Es besteht gegenwärtig keine Möglichkeit, eine Absolute Adressierung für Ausdrücke <256 zu provozieren. Die Adressierungsart wird vom Assembler im 1. Durchlauf bestimmt. Für alle nicht auflösbaren Ausdrücke wird dabei *immer* ein Wert >255 angenommen, wenn beispielsweise ein LABEL noch nicht in der Tabelle gefunden wurde. Mit LABELn gekennzeichnete Zero-Page-Adressen müssen deshalb **immer** am Beginn des Programms mit Hilfe des EQU-Pseudo-Opcodes definiert werden, um fatale Fehler im 2. Durchlauf zu verhindern. ASSESSOR zeigt solche Fehler nicht an, der erzeugte Code ist aber „Müll", da alle folgenden Branch-Befehle und internen JMP's und JSR's falsche Adressen aufweisen.

Der Kommando-Modus

Wenn Sie ASSESSOR erfolgreich gestartet haben, befinden Sie sich im Kommando-Modus. Dieser ist durch das normale Applesoft Prompt-Zeichen Ü oder] und den blinkenden Cursor gekennzeichnet. Tatsächlich reagiert der Apple jetzt auch auf alle Applesoft- und DOS 3.3-Befehle. Sie sollten jedoch außer den im folgenden aufgeführten Befehlen keine anderen ausprobieren, da dies zu unangenehmen Nebenwirkungen führen kann. Im schlimmsten Fall „hängt sich der Apple auf". Insbesondere „FP", „INT" und „FRE" sollten Sie meiden.

NEW

NEW löscht den im Speicher befindlichen Quellcode. Beim Start von ASSESSOR wird automatisch ein NEW durchgeführt.

HOME

HOME dient ganz normal zur Löschung des Bildschirms wie unter Applesoft.

Die folgenden drei Befehle führen Sie aus dem Kommando-Modus in den Editier-Modus. Obwohl die Wirkung je nach Einstieg verschieden ist, steht Ihnen im Editier-Modus immer der gleiche Befehlsvorrat zur Verfügung, der im nächsten Abschnitt besprochen wird.

&A

„&A<RETURN>" ist der „normale" Einstieg in den Editier-Modus und erlaubt die Eingabe von Quellcode. Der neue Text wird grundsätzlich an das Ende eines bereits vorhandenen Textes angefügt (ADD-Funktion). Der Editor enthält eine vollautomatische Zeilennummernausgabe, so daß Sie sich nie um die Zeilennummer zu kümmern brauchen. Die kleinste Zeilennummer ist 1, die größte theoretisch 65535 (vorher ist der Speicher voll). Der Abstand ist fest mit 1 vorgegeben. Ist noch kein Quellcode im Speicher, wird die Zahl 1 ausgegeben und der Cursor dahinter positioniert. Befindet sich dagegen schon Text im RAM, wird die nächste freie Zeilennummer generiert. Bei jedem Aufruf des Editors mit „&A" findet eine Neunumerierung statt (RENUMBER-Funktion) für den Fall, daß Sie mit „DEL" oder auf andere Weise zwischenzeitlich Zeilen gelöscht haben.

&I

Wollen Sie in einen bestehenden Text noch Zeilen einfügen, so rufen Sie den Editier-Modus über „&I*Zeilennummer*“ auf (INSERT-Funktion). „*Zeilennummer*“ gibt dabei die Zeilennummer an, die die erste einzufügende Zeile erhalten soll. Die „*Zeilennummer*“ darf nicht kleiner als 1 sein und muß kleiner sein als die höchste bisher vergebene Zeilennummer, ansonsten erfolgt eine Fehlermeldung. Nach der Eingabe einer Zeile bleiben Sie in der Einfüge-Funktion, so daß Sie nacheinander mehrere Zeilen einfügen können. ASSESSOR schiebt automatisch alle folgenden Zeilen weiter hoch und numeriert sie neu. Sie können die Einfüge-Funktion nur verlassen, indem Sie die Eingabe einer Zeile mit „CTRL-X“ abbrechen oder beim Erscheinen einer neuen Zeilennummer nur <RETURN> eingeben. Diese Zeilennummer wird dann verworfen, so daß keine Lücke bleibt.

&

Um Fehler in bestehenden Zeilen zu berichtigen, rufen Sie den Editiermodus mit „&*Zeilennummer*“ auf. Die gewünschte Zeile wird gelistet und der Cursor auf die erste Position gestellt. Es stehen alle Editierkommandos zur Verfügung. Wenn Sie die Bearbeitung der Zeile abgeschlossen haben, wird automatisch die nächste Zeile zur Korrektur gelistet. Sind keine weiteren Zeilen mehr vorhanden, gelangen Sie in die Anfüge-Funktion (identisch mit „&A“). Die Korrektur-Funktion können Sie mit „CTRL-X“ abbrechen. Die gerade gelistete Zeile bleibt unverändert erhalten.

&L

Dieser Befehl listet das im Speicher befindliche Quellprogramm in formatierter Weise. Wenn Sie vorher mit „PR#*Slot*“ den Drucker eingeschaltet haben, können Sie auch das Listing ausdrucken.
Die Syntax ist etwas vielfältiger:

&L<RETURN>	listet den ganzen Code
&La<RETURN>	listet nur die Zeile a
&La,<RETURN>	listet von Zeile a bis zum Ende
&La,b<RETURN>	listet von Zeile a bis Zeile b einschließlich

Mit „CTRL-S“ können Sie das Listing anhalten und mit einem erneuten Tastendruck wieder starten, „CTRL-C“ bricht den Listvorgang ab (BREAK).

DEL

Mit DEL können Sie einzelne oder auch mehrere zusammenhängende Zeilen des Quellprogramms löschen. DEL verlangt zwei durch Komma getrennte Parameter: die erste zu löschende Zeile und die letzte.
„DEL12,15“ löscht die Zeilen 12 bis 15 inclusive. „DEL3,3“ löscht nur die Zeile 3. Es ist allerdings bequemer, einzelne Zeilen wie in Applesoft nur mit „Zeilennummer <RETURN>“ zu löschen, z.B. „20<RETURN>“ für Zeile 20.

CATALOG

Dieser Befehl gestattet Ihnen, den Inhalt einer Diskette anzusehen. Es sind alle Parameter wie unter Applesoft erlaubt, z.B. „CATALOG,D2“.

LOAD

Mit LOAD können Sie einen vorher abgespeicherten Quellcode-File wieder in den Speicher laden. ASSESSOR-Files werden wie Applesoft-Files gespeichert und haben beim CATALOG-Befehl den Kennbuchstaben „A“, der hier für „ASSESSOR“ steht. Es sind alle normalen Parameter erlaubt, z.B. „LOAD DRUCKROUTINE,D2“.
Wichtig: Auch wenn ASSESSOR-Files äußerlich wie Applesoft-Files wirken, so können sie doch nie von Applesoft verstanden werden. Als kleines (ungefährliches) Experiment können Sie einmal „LIST“ eingeben, um die Applesoft-Interpretation Ihres Quellcodes zu sehen. Versuchen Sie nie, einen ASSESSOR-File mit „RUN“ in den Speicher zu holen.
Bei jedem LOAD wird ein eventuell schon im Speicher befindlicher Quellcode gelöscht, da automatisch ein „NEW“ erfolgt.

SAVE

Mit SAVE und den DOS-üblichen Parametern speichern Sie Ihren Quellcode als Typ-A File ab. Auf der Diskette gleicht er einem Applesoft-File und trägt denselben Kennbuchstaben „A", obwohl er im Inneren völlig anders kodiert ist.

&E

Wenn Sie normale DOS-Textfiles einlesen wollen, die einen Assembler-Quellcode **ohne** Zeilennummern enthalten, verwenden Sie diesen Befehl. (Das „E" steht für „EXEC") Die genaue Syntax lautet:

&E"Filename"<RETURN>

Achten Sie darauf, daß der Filename in Anführungszeichen steht. ASSESSOR liest dann den benannten Textfile ein und kodiert ihn sofort. Dabei sehen Sie in inverser Schrift die von ASSESSOR verteilten Zeilennummern und in normaler Schrift den Inhalt des Files. Ist der gesamte File eingelesen, wird nur eine Zeilennummer ausgegeben, und es passiert nichts mehr, bis Sie „CTRL-X" eingeben (Control-Taste und den Buchstaben X gleichzeitig drücken).

Bei diesem Befehl wird ein eventuell schon im Speicher befindlicher Quellcode **nicht** gelöscht, sondern der Inhalt des Textfiles wird hinten an den alten Text angehängt (APPEND-Funktion). So ist es bequem möglich, oft benutzte Standardroutinen als Textfiles auf der Diskette zu haben und in neue Progamme einzufügen.

Sollte eine Zeile nicht der ASSESSOR-Syntax entsprechen, erhalten Sie eine Fehlermeldung und der Lesevorgang wird abgebrochen. Dies tritt besonders dann auf, wenn Sie Textfiles anderer Assembler wie BIG MAC, LISA oder Merlin lesen, da diese alle etwas unterschiedliche Befehle (speziell bei den Pseudo-Opcodes) benutzen. Sie müssen dann diese Unterschiede mit einem Textfile-Editor (z.B. Applewriter) beheben, bevor Sie den File erneut laden.

Beim Lesen von Textfiles, die mit ASSESSOR selbst erstellt worden sind, sollten keine Probleme auftauchen außer bei DOS-Fehlern (z.B. I/O ERROR), die ebenfalls angezeigt werden und den Leseversuch abbrechen.

&T

Mit diesem Befehl („T" steht für „Textfile") können Sie Ihren Quellcode als normalen DOS-Textfile auf die Diskette schreiben. Die Syntax:

&T"Filename"<RETURN>

Auch hier muß der Filename in Anführungszeichen stehen. ASSESSOR schreibt jetzt eine unformatierte Kopie Ihres Quellcodes ohne Zeilennummern, wie er von vielen anderen Assemblern „verstanden" werden kann oder wie er zur Modem-Übertragung (ASCII-Datei) notwendig ist. Es entsteht ein sequentieller Textfile.
Dieser Befehl ist besonders dazu geeignet, sich kleinere Standardroutinen als Module auf Diskette zu schreiben, die dann bei Bedarf mit dem APPEND-Effekt des &E-Befehls in Ihr neues Programm eingefügt werden, ohne daß Sie die Zeilen noch einmal tippen müssen.

DELETE

Mit DELETE können Sie wie normal Files auf der Diskette löschen. Es sind alle Parameter erlaubt.

LOCK, UNLOCK

Auch diese beiden DOS-Befehle arbeiten wie normal.

PR#

PR#*Slot* leitet die Ausgabe des Apple zu dem bezeichneten Slot. Dieser Befehl dient dazu, einen Drucker zu aktivieren. Falls dessen Interfacekarte in Slot 1 steckt, wird er mit „PR#1" aktiviert und durch „PR#0" wieder ausgeschaltet. Die Interfacekarte muß die Ausgabe auf dem 40-Zeichenschirm „echoen", um eine formatierte Ausgabe zu gewährleisten. Das ist fast immer gegeben.
Schaltet Ihr Interface den Bildschirm ab, erreichen zwar alle Zeichen den Drucker, der Tabulator von ASSESSOR funktioniert aber nicht mehr. Schalten Sie mit PR#*Slot* keine 80-Zeichenkarte ein, da Treiberroutinen dafür nicht in ASSESSOR enthalten sind.

&F

Nach &F wird der noch freie Speicher zwischen dem Ende Ihres Quellcodes und HIMEM: (s.u.) in Bytes angezeigt. Bedenken Sie, daß dieser Wert nicht zu klein werden darf, da bei der Assemblierung auch noch die LABEL-Tabelle hier ihren Platz finden muß. Jedes LABEL, das in der 1. Zeilenposition steht und dadurch definiert wird, benötigt 1 Byte pro Zeichen (max. also 8 Bytes) **plus** 2 Bytes für die Adresse.

HIMEM:

Mit HIMEM: legen Sie die Obergrenze fest, bis zu der Ihr Quellcode **und** die LABEL-Tabelle reichen dürfen. Beim Start vom ASSESSOR liegt diese Grenze bei 29952 ($7500). Vergessen Sie hinter HIMEM: den Doppelpunkt nicht. Die folgende Zahl muß immer **dezimal** sein. Die „Fragezeichen-Funktion" von ASSESSOR hilft Ihnen bei der Umrechnung.
Erhalten Sie bei der Eingabe von Quellcode oder im 1. Durchlauf des Assemblers die Fehlermeldung „? OUT OF MEMORY ERROR", so sind Quellcode und/oder LABEL-Tabelle zu lang. Sie können dann HIMEM: etwas höher setzen, verringern dadurch aber den Platz für den erzeugten Maschinencode (Objektcode). *Setzen Sie HIMEM: niemals über 33952 ($84A0), da dann ASSESSOR selbst überschrieben werden kann.* Erhalten Sie erst im 2. Durchgang des Assemblers die Fehlermeldung „? OUT OF MEMORY ERROR", so reicht der Platz für den Objektcode nicht aus und Sie sollten HIMEM: erniedrigen. Bei extrem langen Programmen setzen Sie zunächst HIMEM: auf 33950 hoch und speichern den fertigen Quellcode ab. Dann setzen Sie HIMEM: wieder herunter und benutzen zur Assemblierung eine „abgespeckte" Version ohne Kommentare, um Speicherplatz zu sparen. Für diesen sehr seltenen Fall hilft Ihnen das Programm KOMMEX (siehe dort), aus Ihrem Quellcode automatisch alle Kommentare zu entfernen. Ohne Kommentare sind Programme mit bis zu 1500 Zeilen normalerweise kein Problem. Bewahren Sie die volle Version zur Dokumentation auf!

&"

ASSESSOR besitzt eine eingebaute Suchfunktion. Sie gestattet es, eine Zeichenfolge im gesamten Quellcode zu suchen, wobei auch Wortteile gefunden werden. Lediglich nach MNEMONICs kann nicht gesucht werden. Der Suchbegriff muß hinter dem Ampersand & von Anführungszeichen eingeschlossen stehen (z.B. &"LOOP"<RETURN>). Alle Zeilen, die den Suchbegriff enthalten, werden gelistet. So ist es einfach möglich, in größeren Programmen zu überprüfen, ob ein LABEL schon benutzt wurde oder aus welchen Programmteilen eine Unterroutine aufgerufen wird.

&ASM

Diese Funktion des Kommando-Modus ist für Ihr Ziel eigentlich die wichtigste: der Aufruf des Assemblers. Der Quellcode muß sich bereits im Speicher

befinden. ASSESSOR zeigt den Ablauf seiner Arbeit für Sie an. Im 1. Durchlauf wird für jede bearbeitete Zeile ein Punkt ausgegeben, im 2. Druchlauf wird der erzeugte Maschinencode mit den zugehörigen Adressen und der Quellcode formatiert gelistet, wenn Sie diese Funktion nicht mit dem Pseudo-Opcode „NLS" ausgeschaltet haben (siehe dort). Ist ein Drucker aktiviert, können Sie das Ergebnis Ihrer Arbeit auch schwarz auf weiß festhalten. Mit Hilfe des „ASSESSOR-FILER"-Programms wird auch direkt in einen Textfile „assembliert", z.B. für Veröffentlichungen. Jeder Fehler im 1. oder 2. Durchlauf wird mit der zugehörigen Zeilennummer angezeigt, und die Assemblierung wird abgebrochen.

&D

Der Assembler erzeugt im 1. Durchlauf eine Tabelle aller LABEL mit den zugehörigen Adressen. Mit „&D" können Sie sich diese Tabelle ausdrucken lassen. Auf dem Bildschirm werden 2 Label pro Zeile ausgegeben, auf einem Drucker 4 Label bei 80 Zeichen Breite. Die Auflistung kann mit „CTRL-S" unterbrochen und mit „CTRL-C" abgebrochen werden (wie beim List-Befehl „&L").
Die Reihenfolge der LABEL entspricht ihrem Auftreten im Programm. Für eine alphabetische Liste müßten Sie Ihr eigenes Sortierprogramm schreiben und über die USER-Funktion „&U" aufrufen.

&S

Der durch die Assemblierung erzeugte Objektcode (Maschinenprogramm) wird mit „&S"Filename" als Binärfile abgespeichert. Sie brauchen weder Startadresse noch Länge zu spezifizieren, da ASSESSOR diese für Sie einsetzt. Durch eine Manipulation des DOS wird dabei nicht die tatsächliche Lage im Speicher des Apple in den File eingetragen, sondern die von Ihnen mit dem Pseudo-Opcode „ORG" gewünschte Adresse. Dadurch können Sie den Binärfile (entsprechende Programmierung vorausgesetzt) direkt über „BRUN" auf dem richtigen Speicherplatz starten.

&?

Die Fragezeichen-Funktion wurde zu Ihrer Unterstützung eingebaut. Sie ermöglicht den direkten Zugriff auf die Rechenroutinen von ASSESSOR. Für die Eingabe gelten die gleichen Regeln wie für einen Ausdruck im OPERANDEN.

Falls die LABEL-Tabelle durch eine vorherige Assemblierung schon Daten enthält, können Sie auch auf diese zugreifen. Allerdings muß das LABEL, mit einem vorangestellten Anführungszeichen, an erster Position im Ausdruck stehen. Mögliche Aufrufe sind

```
&?12345
&?$FF59
&?%101010101
&?"LABEL (nur ein Anführungszeichen !!!)
&?"START+$500-300-%1011010
&?<"LABEL-1
```

Die Operationszeichen + - < > sind erlaubt. Das Ergebnis der Berechnung wird Ihnen hexadezimal, dezimal und binär angezeigt. Liegt das Ergebnis oberhalb von 32767, wird es auch als negative Dezimalzahl ausgegeben, so wie es Applesoft oder Integer Basic als Adresse akzeptieren (z.B. -151). Sie können diese negativen Zahlen jedoch nicht selber eingeben.

CALL-151

Sie verlassen ASSESSOR und gehen in den Apple-Monitor, um sich beispielsweise einige Zeiger von ASSESSOR anzusehen oder einen gerade assemblierten Code. Seien Sie vorsichtig mit Schreiboperationen im Monitor, da Sie dadurch eventuell ASSESSOR zerstören.

&U

Um ASSESSOR offen für spätere Ergänzungen zu lassen, wurde ein spezielles USER-Kommando eingebaut. Nach dem Aufruf durch „&U" springt ASSESSOR nach $03F8, dem Monitor CTRL-Y-Vektor. Wenn Sie diesen Vektor nicht ändern, gelangen Sie genauso in den Apple-Monitor wie mit CALL -151. Sie können allerdings nach $03F8 bis $03FA einen eigenen Sprungvektor (JMP XYZ) setzen und auf diese Weise bequem in einen zusätzlichen Programmteil verzweigen. Das Programm „ASSESSOR-FILER" zeigt, wie Sie noch weitere Parameter mit übergeben können.

Der Editier-Modus

ASSESSOR besitzt einen eingebauten Zeileneditor, der Ihnen die Arbeit wesentlich erleichtert. Sie erreichen ihn aus dem Kommando-Modus über die Befehle „&A", „&I*Zeilennummer*" und „&*Zeilennummer*". Der Cursor steht nach dem Aufruf immer auf der 1. Eingabeposition. Sie sollten niemals versuchen, vor dieser Position etwas zu verändern, auch wenn Sie mit den Pfeiltasten bis zur Zeilennummer gelangen können. Folgende Kommandos sind wirksam:

→ (Pfeil nach rechts)
Der Cursor bewegt sich um eine Position nach rechts.

← (Pfeil nach links)
Der Cursor bewegt sich um eine Position nach links.

<ESC> → und <ESC> ←
Für schnelle Cursorbewegungen steht Ihnen die Express-Funktion zur Verfügung. Wenn Sie vor der Pfeiltaste die <ESCAPE>-Taste drücken, bewegt sich der Cursor mit hoher Geschwindigkeit in die gewünschte Richtung, bis er das Zeilenende erreicht oder über der Zeilennummer steht. Sie können die Bewegung jederzeit stoppen, indem Sie eine Taste drücken.

<CTRL>-B
Der Cursor wird auf die 1. Eingabeposition gesetzt (Beginn).

<CTRL>-N
Der Cursor wird auf die letzte Position der Zeile gesetzt (hinter dem letzten eingegebenen Zeichen = eNde). „B" und „N" liegen auf der Tastatur in der „richtigen" Anordnung nebeneinander.

<CTRL>-D
Das Zeichen, auf dem sich der Cursor gerade befindet, wird gelöscht, alle folgenden Zeichen rücken um eine Position nach links.

<CTRL>-I
Ab der augenblicklichen Cursorposition werden Zeichen in die Zeile eingefügt, die folgenden Zeichen (einschließlich des Zeichens unter dem Cursor) nach rechts geschoben. Dieser Modus bleibt auch nach der Eingabe eines Zeichens bestehen und wird durch ein anderes CTRL-Zeichen oder eine der beiden Pfeiltasten beendet. Eine Zeile kann maximal 239 Zeichen (incl. Zeilennum-

mer) fassen. Sie sollten in Assembler-Quellcodes so lange Zeilen jedoch nicht benutzen, da dann keine einwandfreie Formatierung mehr möglich ist.

<CTRL>-X
Der Editier-Modus wird durch diesen Befehl verlassen. Die gerade bearbeitete Zeile verbleibt unverändert im Speicher, falls sie dort schon existierte (Aufruf mit „*&Zeilennummer*") oder wird ignoriert. Sie können den Editier-Modus auch verlassen, wenn Sie bei einer leeren Zeile (nur Zeilennummer) ein <RETURN> eingeben.

<CTRL>-R
Falls Sie einmal zuviel an einer Zeile editiert haben, gibt „CTRL-R" die ursprüngliche Zeile erneut aus, wenn diese schon im Speicher existiert (Aufruf mit „*&Zeilennummer*"). War die Zeile noch nicht abgespeichert, so hat dieser Befehl keine Wirkung.

<RETURN>
Mit <RETURN> wird die gesamte Zeile in den Speicher übernommen, unabhängig von der momentanen Cursorposition (im Gegensatz zu Applesoft!). ASSESSOR generiert daraufhin die nächste Zeilennummer. Wurde <RETURN> bei einer leeren Zeile eingegeben, wird der Editier-Modus aufgehoben.

<CTRL>-Q
Wollen Sie die Zeile nur bis zur Cursorposition übernehmen (wie bei Applesoft), geben Sie „CTRL-Q" ein. Das Zeichen unter dem Cursor und alle folgenden werden ignoriert.

Die Pseudo-Opcodes von ASSESSOR

Um den Assemblierungsprozess zu beeinflussen und um Daten in Ihrem Programm zu definieren, besitzt ASSESSOR eine Reihe von Pseudo-Opcodes. Sie reservieren Platz, legen Adressen fest, bestimmen den Speicher für den entstehenden Objektcode oder steuern Eigenschaften von ASSESSOR. Lediglich zwei dieser Zusatzkommandos werden in echte Maschinenbefehle übersetzt: „BGE" und „BLT", die Synonyme für „BCS" und „BCC" darstellen und nur zu Ihrer Bequemlichkeit eingeführt wurden, da sie in vielen Fällen anschaulicher sind (s.u.). Die übrigen 13 Pseudo-Opcodes weisen nur ASSESSOR an,

was zu tun ist, und sind nicht für den Microprozessor bestimmt. Alle Pseudo-Opcodes stehen grundsätzlich im MNEMONIC-Feld!

ORG

Syntax: ORG Ausdruck

Der Assembler übersetzt Ihren Quellcode in ablauffähigen Maschinencode, den Objektcode. Dieser muß irdendwo gespeichert werden. Aus technischen Gründen wird dies immer an der selben Stelle im Hauptspeicher des Apple geschehen, und zwar unmittelbar oberhalb von HIMEM:. Sie können diese Position nur in geringem Maße durch die Verlegung von HIMEM: beeinflussen. Normalerweise verändern Sie HIMEM: nicht von seinem Standardwert $7500. Um Ihnen trotzdem die Möglichkeit zu geben, Ihr Programm überall im RAM ablaufen zu lassen, wurde das Pseudo-Opcode „ORG" (= ORiGin, Ursprung) geschaffen. Es setzt für das Programm den Anfangsort fest, an dem es **ausgeführt** wird. In den meisten Fällen dürfte diese Adresse eine andere sein als die, an der der Code **erzeugt** wird. Nur wenn Sie keinen „ORG" definieren, sind beide Adressen identisch. Das Pseudo-Opcode „ORG" sollte der erste Befehl in *jedem* Assemblerprogramm sein, da er die Adressen aller internen Sprünge beeinflußt. Der Befehl

```
ORG $300
```

spezifiziert z.B. den Programmstart mit $300 (768 dezimal).
Achten Sie darauf, daß der von Ihnen gewünschte Speicher auch für das Maschinenprogramm zur Verfügung steht. Ein „ORG $400" z.B. würde Ihren Textbildschirm ganz schön durcheinander bringen, ein „ORG $C000" mit Sicherheit den „Absturz" des Apple verursachen.
Damit Sie nicht mühselig den Objektcode von seinem Erzeugungsort zum Ausführungsort verschieben müssen, bevor Sie ihn abspeichern können, „schmuggelt" ASSESSOR die von Ihnen mit „ORG" gewünschte Startadresse in den Binärfile, wenn Sie den Objektcode mit dem Befehl „&S"Filename" abspeichern. Ein folgendes „BLOAD" oder „BRUN" läd den File immer nach „ORG".
In einem Programm können mehrere „ORG"-Befehle auftauchen. Diese Möglichkeit ist **nur** für fortgeschrittene Assemblerprogrammierer gedacht, da damit viele Probleme verbunden sind. Auch bei mehreren „ORG"s wird grundsätzlich ein im Speicher zusammenhängender Code erzeugt, der unter der Adresse des ersten „ORG"s auf Diskette gespeichert wird. Alle internen Adressen werden jedoch stets durch den letzten wirksamen „ORG"-Befehl definiert. Sie sind

selbst dafür verantwortlich, daß nach dem Laden des Files alle Teile Ihres „Multi-ORG"-Programms auf die vorgesehenen Speicherstellen verschoben werden. Anwendungen treten z.B. bei RAM-Disk-Treibern und anderen Programmen auf, die in der Language Card laufen sollen. Benutzen Sie mehr als einen „ORG" pro Programm nur, wenn es nicht anders geht.

EQU

Syntax: Label EQU Ausdruck

„EQU" (= EQUates, ist gleich) wird benutzt, um den Wert eines LABELs zu definieren, in der Regel eine externe Adresse (z.B. im ROM) oder eine Konstante, der ein sinnvoller Name zugeordnet werden soll. Es ist empfehlenswert, alle „EQU" in einem Definitionsteil gleich zu Beginn des Programms anzuordnen. Wenn mit „EQU" eine Zero-Page Adresse oder ein 1-Byte Wert vereinbart werden soll, **muß** dieses vor der ersten Nutzung des Labels geschehen, um schwerwiegende Fehler bei der Übersetzung zu vermeiden. Mögliche Anwendungen sind:

```
COUT   EQU   $FDED
CSWL   EQU   $36
CSWH   EQU   CSWL+1
```

Enthält der Ausdruck nach „EQU" selbst ein LABEL, so muß dieses vorher definiert worden sein, um die Fehlermeldung „MISSING LABEL" zu vermeiden. Das obige Beispiel ist also korrekt.
Jedes LABEL darf nur einmal im Programm definiert sein. Dagegen darf jede Adresse beliebig oft mit einem unterschiedlichen LABEL versehen werden:

Falsch:
```
OUTPUT EQU $FDED
OUTPUT EQU $FDF0
```

Erlaubt:
```
COUT1 EQU $FDF0
PRINT EQU $FDF0
```

ASC

Syntax: ASC "String"
ASC 'String'

Das Pseudo-Opcode „ASC" (= ASCII) weist ASSESSOR an, den folgenden Text, beginnend ab der augenblicklichen Position, direkt in den Speicher zu schreiben. Ist der Text von Anführungszeichen eingeschlossen, wird in jedem Byte das Bit 7 gesetzt. Hochkommata bewirken die Löschung von Bit 7.

ASC "ABC" erzeugt $C1 $C2 $C3
ASC 'ABC' erzeugt $41 $41 $43

In der eingeschlossenen Zeichenkette dürfen selber Anführungszeichen oder Hochkommata vorkommen. Erlaubt ist also:
ASC "Er sagte: "Guten Tag""
Es gibt drei Ausnahmen: Nach einem in den String eingefügten Hochkomma oder Anführungszeichen darf als nächstes Zeichen **nicht** ein „+", „-" oder Leerzeichen folgen, da dann ASSESSOR ein Stringende annimmt.
Die Länge des eingeschlossenen Strings darf theoretisch 255 Zeichen nicht überschreiten (Mit dem Editor können Sie soviele Zeichen gar nicht eingeben!).

INV

Syntax: INV "String" oder INV 'String'

„INV" (= INVerse) speichert den folgenden Text in dem Format ab, der auf dem Bildschirm des Apple inverse Zeichen erzeugt. Der Text muß von Anführungszeichen oder Hochkommata eingeschlossen sein, die allerdings keinen Einfluß auf das Format haben. Es gelten dieselben Einschränkungen wie bei „ASC".

FLS

Syntax: FLS "String" oder FLS 'String'

„FLS" (= FLaSh) ist identisch mit „INV", nur werden blinkende Zeichen erzeugt.

DCI

Syntax: DCI "String"
DCI 'String'

„DCI" (= Declare Characters Immediate) ist identisch mit „ASC", nur weist das letzte Zeichen im String genau das umgekehrte Bit 7 auf wie die voranstehenden Zeichen. Dieses Pseudo-Opcode eignet sich besonders zur Erstellung von Tabellen.

DCI "ABC" erzeugt $C1 $C2 $43
DCI 'ABC' erzeugt $41 $42 $C3

HEX

Syntax: HEX Hexadezimale Daten

„HEX" (= HEXadezimal) erlaubt Ihnen, hexadezimale Daten oder Konstanten für Tabellen, Arrays oder als Begrenzungen in Ihr Programm einzufügen. Abweichend von den übrigen Regeln darf hier **kein „$"** vor den Werten stehen. Der Ausdruck muß aus hexadezimalen Zahlen mit jeweils zwei Ziffern bestehen, also „0A" und nicht einfach „A". Diese Zahlen können direkt aneinander geschrieben werden. Wenn die Summe der Ziffern ungerade ist oder Sie mehr als 128 Hex-Zahlen eingeben, erfolgt eine Fehlermeldung. Als alternative Schreibweise können Sie nach jeder Zahl ein Komma setzen. In diesem Fall können Sie ausnahmsweise auch einziffrige Zahlen schreiben.

HEX 8D8787F0DE	erlaubt
HEX 8D0787434	nicht erlaubt, da ungerade
HEX 8D,7,C1,C2	erlaubt
HEX 8D,7,C1C2	nicht erlaubt, da Schreibweisen gemischt
HEX $8E	nicht erlaubtes „$"

ADR

Syntax : ADR Ausdruck

„ADR" (= ADResse) speichert den 2-Byte Wert des Ausdrucks, der normalerweise eine Adresse ist, in zwei aufeinanderfolgenden Bytes des Objektcodes. Wie beim 6502-Prozessor üblich, wird zuerst das niederwertige, danach das höherwertige Byte abgelegt.

```
ADR $E003 ergibt $03 $0E
ADR $FF ergibt $00 $FF
```

Mit „ADR" können Sie besonders einfach Sprungtabellen aufbauen, wenn Sie von den Rechenfertigkeiten des Assemblers Gebrauch machen.

```
ADR ZIEL1-1
ADR ZIEL2-1
ADR ZEIL3-1
  usw.
```

DFB

Syntax: DFB < Ausdruck
DFB > Ausdruck

DFB (DeFine Byte, bestimme ein Byte) erzeugt entweder den höherwertigen oder den niederwertigen Anteil des Ausdrucks und legt ihn im Objektcode ab.

```
DFB < $1234 ergibt $34
DFB > $1234 ergibt $12
```

DFS

Syntax: DFS Ausdruck

Gelegentlich brauchen Sie im Code freien Platz, um Raum für Zwischenspeicherungen z.B. eines Namens, einer Adresse oder auch eines einzelnen Wertes zu

haben. „DFS“ (= DeFine Storage, reserviere Speicherplatz) generiert keinen Code, sondern schafft freien Raum, indem es den folgenden Code im Speicher um die mit *Ausdruck* bezeichnete Anzahl von Bytes weiterschiebt. Der Wert von *Ausdruck* darf nicht negativ sein und muß theoretisch kleiner als 65536 sein. Soviel Platz ist an einem Stück aber gar nicht im Apple vorhanden. Enthält der Ausdruck ein LABEL, so muß dieses vorher definiert worden sein, damit ASSESSOR den benötigten Platz richtig berechnet.

END

Syntax: END

Dieses Pseudo-Opcode (= ENDe) bedeutet, daß der noch folgende Rest des Quellcodes ignoriert werden soll. LABEL nach „END“ werden nicht in die LABEL-Tabelle eingetragen und können im Programm vor „END“ nicht benutzt werden. Ihr Programm muß nicht, wie ein BASIC-Programm, mit „END“ aufhören, da ASSESSOR das Ende selbsttätig erkennt. „END“ dient lediglich in Testphasen dazu, Teile des Programms zu assemblieren, ohne daß der gesamte Rest gelöscht werden muß.

LST, NLS

Syntax: LST
NLS

Diese beiden Befehle kontrollieren, ob das Assemblerlisting im 2. Durchlauf auf dem Bildschirm und auf einem aktivierten Drucker ausgegeben wird oder nicht. Normalerweise ist die List-Option eingeschaltet.
Mit „NLS“ (= No LiSting) wird ab der betreffenden Zeile die List-Option ausgeschaltet. „LST“ (= LiST) aktiviert die List-Option wieder. Beide Befehle zusammen ermöglichen es Ihnen, nur ausgewählte Teile, z.B. ein verändertes Unterprogramm, zu betrachten, indem Sie in Zeile 1 ein „NLS“, vor die interessierende Routine ein „LST“ und danach wieder ein „NLS“ eintragen.
Auch lange Programme, von denen Sie wissen, daß sie weitgehend fehlerfrei sind, können Sie vorteilhaft ohne Listing assemblieren, da dies die Ausführungsgeschwindigkeit bei vielen Kommentaren leicht um den Faktor 10 steigern kann. Der eigentliche Übersetzungsvorgang wird durch „LST“ oder „NLS“ nicht verändert.

BLT, BGE

Diese beiden Befehle sind im strengen Sinne keine Pseudo-Opcodes sondern lediglich Synonyme für richtige MNEMONICs. Sie sollen Ihnen helfen, nach Vergleichsoperation (CMP, CPY, CPX) leichter den richtigen Befehl zu finden. „BLT" steht für „Branch on Lower Than" (= Verzweige, wenn kleiner) und erzeugt den selben Maschinencode wie „BCC" (Branch on Carry Clear), was aber weniger einprägsam ist. „BGE" bedeutet „Branch on Greater than or Equal" (Verzweige, wenn größer oder gleich) und erzeugt den selben Code wie „BCS" (Branch on Carry Set). „BLT"<=>„BCC" und „BGE"<=>„BCS" sind völlig gleichwertig und es bleibt Ihnen überlassen, welchen Befehl Sie verwenden.

Fehlermeldungen

Um Speicherplatz zu sparen, benutzt ASSESSOR einen Teil der Fehlermeldungen des Applesoft mit. Um in kein Sprachwirrwarr zu geraten, wurden die zusätzlichen Nachrichten auch in englischer Sprache ausgeführt.

„? SYNTAX ERROR"
a) Kommando-Modus: nicht erkanntes oder unvollständiges Kommando, besonders wenn Sie ein Anführungszeichen vergessen haben.
b) Editier-Modus: ungültiges MNEMONIC oder, wenn kein LABEL vorhanden ist, das Leerzeichen zu Beginn der Zeile vergessen.
c) 1. Durchlauf des Assemblers: fehlerhafter OPERAND.

„? ILLEGAL QUANTITY ERROR"
a) Kommando-Modus: Es ist nicht möglich, Zeilen hinter der letzten Zeile „einzufügen".
b) Fragezeichen-Funktion oder Assemblierung: ein Wert ist kleiner als Null oder größer als 65535. Ebenso erhalten Sie im Assembler diese Fehlernachricht, wenn nur ein 1-Byte großer Wert erlaubt ist (z.B. Unmittelbare Adressierung), Ihr Wert aber größer als 255 ist.

„? OUT OF MEMORY ERROR"
a) Im Editor: Der Quellcode paßt nicht in den vorgesehenen Speicher.
b) 1. Durchlauf: Die LABEL-Tabelle paßt zusätzlich nicht mehr unter HIMEM:.
c) 2. Durchlauf: Der Objektcode ist größer als der vorgesehene Platz.
Abhilfe siehe HIMEM:

„? UNDEF'D STATEMENT ERROR"
2. Durchlauf: Diese Meldung erfolgt, wenn Sie eine nicht erlaubte Adressierungsart verwendet haben. Möglicherweise haben Sie bei einer indirekt indizierten Adressierung keine Zero-Page-Adresse angegeben.

„MISSING LABEL"
a) 1. Durchlauf: Nach EQU oder DFS steht ein LABEL, das nicht vorher definiert ist.
b) 2. Durchlauf: Ein LABEL ist nicht gefunden worden.

„DUPLICATE LABEL"
1. Durchlauf: Dieselbe Label-Bezeichnung wurde mehrfach definiert. Die Such-Funktion zeigt Ihnen leicht, wo der Name auftaucht.

„BAD BRANCH"
2. Durchlauf: Die erlaubte Sprungweite für einen Branchbefehl ist überschritten.

Bei Diskettenoperationen können Sie außerdem alle Fehlermeldungen des DOS bekommen.

Besondere Hinweise

Sie sollten ASSESSOR nur über das Applesoftprogramm „START ASSESSOR" starten und nie direkt den Binärfile aufrufen, da dann in den meisten Fällen Teile des Assembler-Moduls zerstört werden.
Der <RESET>-Taste sollten Sie möglichst fern bleiben. Im falschen Moment gedrückt, kann Ihr Quellcode, ASSESSOR oder sogar eine Diskette unbrauchbar werden. Benutzen Sie <RESET> nur, wenn Ihr Apple außer Kontrolle gerät. Das kann z.B. dann geschehen, wenn Sie im Kommando-Modus einen nicht erlaubten Applesoft-Befehl (wie FRE) geben. (Sie haben doch eine Sicherungskopie gemacht?).
Nach den Befehlen „&L" sowie „&D" können Sie das Listing mit <CTRL>-S zeitweilig stoppen und mit <CTRL>-C abbrechen. <CTRL>-S wirkt auch während des 2. Durchlaufs des Assemblers, wenn die List-Option eingeschaltet ist. Betätigen Sie aber **nicht** die <CTRL>-C Taste, da dann der Assembler durcheinander gerät.
Wenn Sie ein Listing mit <CTRL>-S unterbrochen haben, kann die nächste Befehlseingabe zur Meldung „? SYNTAX ERROR" führen, auch wenn Sie

alles richtig gemacht haben. Das ist eine „Eigenart" des Apple und liegt nicht an ASSESSOR. Geben Sie den Befehl erneut, und alles läuft wie gewünscht.
ASSESSOR unterstützt die erweiterten Befehle des 65C02-Prozessors, wie Sie ihn im Apple IIc oder im neuen IIe (ab Sommer 1985) finden. Da ASSESSOR selber diese intern nicht benutzt, ist er auch auf den älteren Geräten voll lauffähig. Auf einem Apple IIplus können Sie so Assemblerprogramme für den 65C02 *schreiben* und *assemblieren*. Diese Programme sind allerdings nur auf Geräten *ausführbar*, wenn diese den 65C02-Prozessor enthalten. Benutzen Sie die neuen Befehle nicht, wenn Ihr Programm auf allen Sorten des Apple eingesetzt werden soll.
ASSESSOR wurde sorgfältig geplant und programmiert. Die endgültige Version wurde unter verschiedenen Bedingungen getestet. Trotzdem ist es bei einem Assemblerprogramm dieser Länge (mehr als 2700 Quellcode-Zeilen) nicht möglich, 100%ige Fehlerfreiheit zu garantieren. Wenn Sie glauben, einen „BUG" gefunden zu haben, benachrichtigen Sie bitte den Autor über die Adresse des Verlages.

Interna

Anmerkung: Die Kenntnis dieses Abschnittes ist für die normale Benutzung von ASSESSOR *nicht* notwendig. Hier sollen lediglich Informationen für den erfahreneren Assemblerprogrammierer (oder für andere wißbegierige Applebesitzer) gegeben werden, die das innere Arbeiten von ASSESSOR verstehen wollen, weil sie z.B. Ergänzungen schreiben oder einige Dinge nach ihrem eigenen Geschmack verändern wollen.
Alle Teile von ASSESSOR sind in dem Binärfile „ASSESSOR" enthalten.
Um möglichst viel freien Benutzerspeicher zu erhalten, wird für DOS „MAXFILES 1" gesetzt (durch das Startprogramm), so daß der einzige verbleibende DOS-Puffer bei $9AA6 beginnt. Unmittelbar darunter liegt ab $84A0 ASSESSOR. Um möglichst wenig Code zu benötigen, macht ASSESSOR ausgiebig von ROM-Routinen im Appelsoft und Autostart-Monitor Gebrauch. Sie sollten für eigene Ergänzungen alle Zero-Page Speicherstellen meiden, die vom Monitor oder von Applesoft benutzt werden (mit Ausnahme der nur für Grafik benutzten Adressen). Zusätzlich belegt ASSESSOR die Speicher $0000 bis $0003, $00E3 sowie $00CE und $00CF. $00FA bis $00FF sind beispielsweise weiterhin frei, ebenso $00EB bis $00EF.
Der gesamte RAM-Bereich des Apple ist belegt (durch DOS, Stack, Tastaturpuffer, Textbildschirm, ASSESSOR, den Quellcode, die LABEL-Tabelle oder

Speicherbelegung unter ASSESSOR

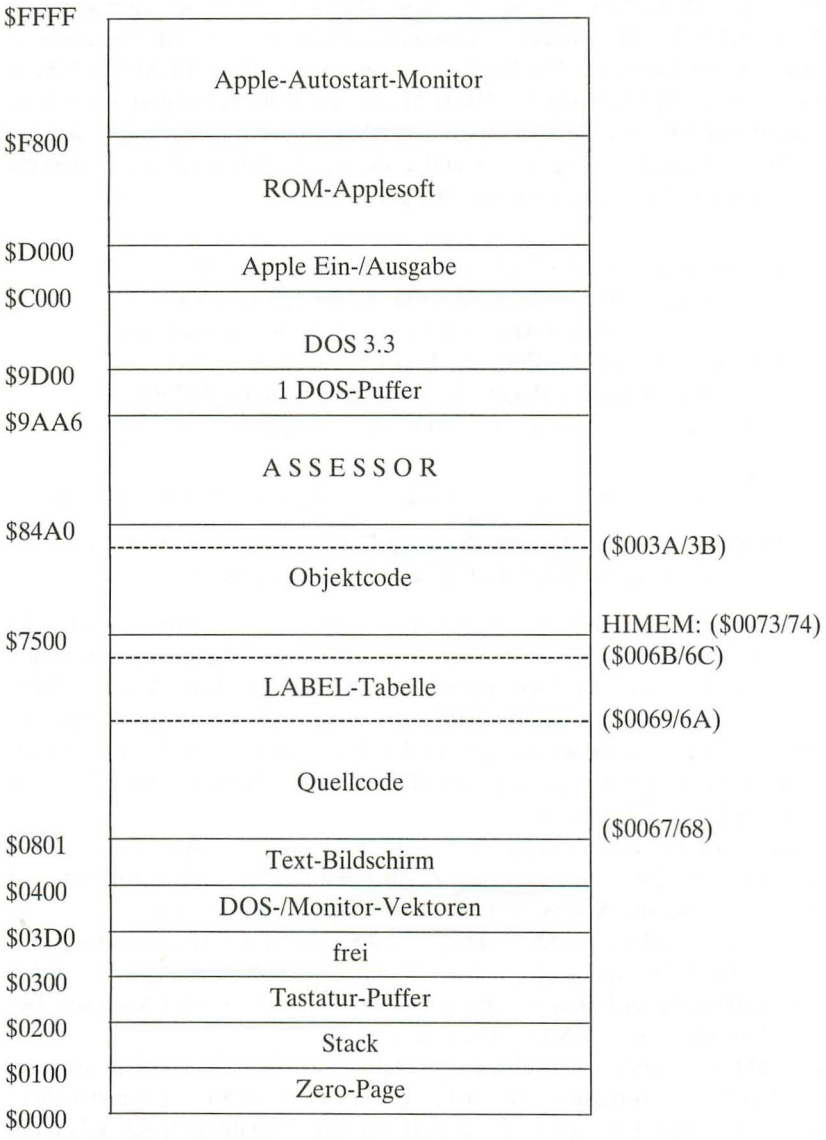


Abb. 37: Speicherbelegung unter ASSESSOR

den Objektcode). Lediglich von $0300 bis $03CF liegt ein kleiner bisher ungenutzter Raum, der Ihnen zur Verfügung steht.
Der Quellcode wird normalerweise von $0801 an aufwärts im Speicher abgelegt. Die LABEL-Tabelle schließt sich unmittelbar daran an, so daß ihre absolute Lage von der Länge des Quellcodes abhängt. Quellcode und LABEL-Tabelle zusammen enden spätestens bei HIMEM:, das auf $7500 voreingestellt ist. Dort beginnt der Objektcode, der bis unter ASSESSOR reichen kann, also bis $84A0. Auf der Zero-Page liegen einige Zeiger, die Ihnen Auskunft über die aktuelle Lage der Grenzen im Speicher geben.

$003A/3B zeigen auf das Ende des Objektcodes (variabel)
$0067/68 zeigen auf den Beginn des Quellcodes (normal $0801)
$0069/6A zeigen auf den Anfang der LABEL-Tabelle (variabel)
$006B/6C zeigen auf das Ende der LABEL-Tabelle (variabel)
$0073/74 zeigen auf den Beginn des Objektcodes (normal $7500)
$00AF/B0 zeigen auf das Ende des Quellcodes (variabel)

Jede Zeile des Quellcodes wird im Speicher in folgendem Format abgelegt:

2-Byte Zeiger auf die nächste Zeile / 2-Byte für die Zeilennummer / der Quelltext, z.T. codiert / 00 als Kennzeichnung des Zeilenendes.

Wenn Sie sich schon einmal näher mit Applesoft beschäftigt haben, werden Sie erkennen, daß dies im Prinzip dem internen Aufbau von Applesoftprogrammen entspricht. Der Text ist jedoch anders als bei Applesoft codiert. Alle MNEMONICs besitzen ein 1-Byte Token, alle zusammenhängenden Leerzeichen einer Zeile (außer in Strings) werden jeweils durch nur ein Token ($FF) dargestellt, das Kommentarzeichen „;“ wird zu $FE. Diese Codierung spart Platz und beschleunigt die Übersetzung.
In der LABEL-Tabelle stehen alle Labelbezeichnungen in ihrer vollen Länge mit gelöschtem Bit 7. Nur das letzte Zeichen hat Bit 7 gesetzt. Sofort dahinter folgt die zugehörige Adresse (2 Byte) im üblichen Lo-Hi-Format.
Im Kommando-Modus ist DOS aktiv, so daß Sie auch DOS-Kommandos geben können. Im Editiermodus werden DOS und Applesoft umgangen, damit Sie auch LABEL-Bezeichnungen verwenden können, die DOS- oder Applesoft-Befehle darstellen wie „TEXT“, „INIT“ usw.
ASSESSOR verändert zeitweilig einige Stellen innerhalb des DOS, macht aber alle „Patches“ selbsttätig wieder rückgängig, es sei denn, Sie würden zwischenzeitlich die <RESET>-Taste drücken. Booten Sie deshalb nach der Arbeit mit ASSESSOR immer neu, besonders wenn Sie Disketten initialisieren wollen.

Hilfsprogramme für ASSESSOR

KOMMEX

Bei einem sehr langen Quellcode kann der zur Verfügung stehende Speicherplatz eventuell nicht ausreichen, und Sie erhalten die Fehlermeldung „? OUT OF MEMORY ERROR". Wenn Ihr Programm viele Kommentare enthält, können Sie den Umfang fast immer soweit reduzieren, daß diese „abgespeckte" Version assembliert werden kann. Um Ihnen die mühevolle Arbeit abzunehmen, alle Kommentare mit dem Editor Zeile für Zeile zu entfernen, wurde KOMMEX entwickelt, das weitgehend vollautomatisch dieses für Sie besorgt.
Um KOMMEX zu benutzen, verfahren Sie folgendermaßen:
1. Speichern Sie Ihren Quellcode auf Diskette ab („SAVE Code"), um eine kommentierte Version zu behalten.
2. Speichern Sie den Quellcode mit dem „&T"-Befehl noch einmal als Textfile ab (anderer Name!).
3. Starten Sie **KOMMEX** mit „BRUN KOMMEX.OBJ".
4. Folgen Sie den Angaben auf dem Bildschirm. Benutzen Sie als „Namen" die Bezeichnung des *Textfiles*.
5. KOMMEX liest den Textfile ein, entfernt alle Kommentare und schreibt ihn zurück, wobei an den alten Namen ein „EX" (= Kommentare Ex) angehängt wird.
6. Starten Sie **ASSESSOR** neu mit „RUN START ASSESSOR". (WICHTIG!)
7. Laden Sie mit dem „&E'-Befehl den erzeugten Textfile.
8. Assemblieren Sie ihn wie gewohnt.

Die „Automatik" von KOMMEX versagt, wenn in Strings (z.B. nach ASC u.a.) ein „;" steht. In diesen (sicher nicht häufigen) Fällen müssen Sie mit dem Editor den abgeschnittenen Rest des Strings wieder ergänzen.
(Am Rande: Wenn Sie sich fragen, wie KOMMEX Ihren Textfile so schnell liest: es wird der „UNLOCK-Trick" von U. Stiehl (Apple DOS 3.3 Tips und Tricks, Seite 127) in leicht veränderter Form benutzt.)

ASSESSOR-Filer

Mit **ASSESSOR-Filer** speichern Sie ein assembliertes Listing als Textfile auf Diskette ab. Dieser enthält sowohl den Quellcode als auch den Maschinencode, so wie Sie ihn im 2. Durchlauf auf dem Bildschirm sehen. Sie können diesen Textfile mit einem Textverarbeitungsprogramm weiter verändern, um ihn z.B. in eine Anleitung mit aufzunehmen, ihn einer Zeitschrift zur Veröffentlichung zuzusenden usw. Auch zu Ihrer eigenen Dokumentation können Sie so „schöne" Listings erzeugen, die Sie mit allen Finessen Ihrer Textverarbeitung (Kopfzeilen, Seitenumbruch usw.) ausstatten können.

Die Bedienung ist denkbar einfach:

1) Starten Sie **ASSESSOR** wie gewohnt und laden Sie den Quellcode.
2) Vergewissern Sie sich mit „&ASM", daß die Assemblierung ohne Fehler erfolgt.
3) Geben Sie im Kommando-Modus „BRUN ASSESSOR.FILER.OBJ" ein.
4) Nach dem Befehl „&U"Filename"" wird auf die Diskette assembliert, wobei an den angegebenen Filenamen noch ein „.L" (= Listing) angehängt wird.

Anmerkung: Wenn Sie den „Applewriter" zum Ausdruck des Listings benutzen wollen, müssen Sie die Punkte nach der Meldung „1. Durchlauf" löschen, da der „Applewriter" durch so viele Punkte durcheinanderkommt (er interpretiert sie als Druckkommandos).

```
1   ;*********************************
2   ; ASSESSOR-FILER für die direkte *
3   ; Assemblierung in ein Textfile  *
4   ;*********************************
5   ;
6   ; von Dr. Jürgen B. Kehrel
7   ;
8             ORG $300
9   ;
10  PROMPT    EQU $33
11  CSWL      EQU $36          ;Ausgabevektor
12  CSWH      EQU $37
13  CHRGET    EQU $B1          ;Zeichen lesen
14  CURLIN    EQU $75          ;aktuelle Zeilennr.
15  TXTPTR    EQU $B8          ;Textzeiger
16  RUNMODE   EQU $D9          ;RUN-Indikator
17  PUFFER    EQU $280         ;Zwischenpuffer
18  DOSWRM    EQU $3D0         ;Warmstart
```

```
                        19  CONNECT   EQU $3EA        ;DOS anhängen
                        20  CTRLY     EQU $3F8        ; = USER
                        21  ASSEMBLE  EQU $84A3       ;ASSESSOR-Aufruf
                        22  SYNTERR   EQU $DEC9       ;Fehlermeldung
                        23  COUT      EQU $FDED       ;Ausgabe
                        24  COUT1     EQU $FDF0       ;Bildschirm
                        25  ;
                        26  ; Initialisiert der USER-Vektor
                        27  ;
0300: D8                28  START     CLD             ;Binärmode
0301: A9 0C             29            LDA #<DISK      ;Programm auf
0303: 8D F9 03          30            STA CTRLY+1     ;den USER-Vektor
0306: A9 03             31            LDA #>DISK      ;legen
0308: 8D FA 03          32            STA CTRLY+2
030B: 60                33            RTS             ;zurück
                        34  ;
030C: 20 B1 00          35  DISK      JSR CHRGET      ;nächstes Zeichen
030F: C9 22             36            CMP #'"'        ;Name in ""
0311: F0 03             37            BEQ DISK1       ;O.K.
0313: 4C C9 DE          38  NAMEERR   JMP SYNTERR     ;Fehler
                        39  ;
0316: A0 01             40  DISK1     LDY #$01        ;Index
0318: B1 B8             41  NAMELOOP  LDA (TXTPTR),Y ;Filenamen lesen
031A: 99 7F 02          42            STA $PUFFER-1,Y ;umkopieren
031D: F0 F4             43            BEQ NAMEERR     ;" fehlt am Ende
031F: C9 22             44            CMP #'"'        ;Ende gefunden?
0321: F0 07             45            BEQ FIOPEN      ;Ja
0323: C8                46            INY             ;Nein, nächstes Z.
0324: C0 1F             47            CPY #31         ;max. 30 für DOS
0326: 90 F0             48            BLT NAMELOOP    ;O.K.
0328: B0 E9             49            BGE NAMEERR     ;zu lang
                        50  ;
032A: 20 EA 03          51  FIOPEN    JSR CONNECT     ;DOS ankoppeln
032D: A9 80             52            LDA #$80        ;Run-Modus
032F: 85 33             53            STA PROMPT      ;herstellen
0331: 85 76             54            STA CURLIN+1
0333: 85 D9             55            STA RUNMODE
0335: A0 1C             56            LDY #$1C        ;MON Output
0337: 20 6B 03          57            JSR PRINT       ;herstellen
033A: A0 00             58            LDY #$00        ;File öffnen
033C: 20 6B 03          59            JSR PRINT
033F: 20 58 03          60            JSR FILE        ;Name ausgeben
0342: A0 07             61            LDY #$07        ;WRITE
0344: 20 6B 03          62            JSR PRINT
0347: 20 58 03          63            JSR FILE        ;wieder Name
034A: A9 9B             64            LDA #<AF        ;alle Ausgaben um-
034C: 85 36             65            STA CSWL        ;lenken durch den
034E: A9 03             66            LDA #>AF        ;ASSESSOR-Filer
0350: 85 37             67            STA CSWH
0352: 20 EA 03          68            JSR CONNECT     ;DOS einstellen
0355: 4C A3 84          69            JMP ASSEMBLE    ;File assembl.
                        70  ;
0358: A0 00             71  FILE      LDY #$00        ;Filenamen aus
035A: B9 80 02          72  FILE1     LDA PUFFER,Y    ;Puffer ausgeben
```

```
035D: C9 22     73             CMP #'"'       ;Ende erreicht?
035F: F0 08     74             BEQ FILE2      ;Ja
0361: 09 80     75             ORA #$80       ;Bit 7 setzen
0363: 20 ED FD  76             JSR COUT       ;ausgeben
0366: C8        77             INY            ;nächstes Zeichen
0367: D0 F1     78             BNE FILE1
0369: A0 18     79   FILE2     LDY #$18       ;.L anhängen
                80   ;
036B: B9 77 03  81   PRINT     LDA TEXT,Y     ;Text ausgeben,
036E: F0 06     82             BEQ ENDE       ;abhängig von Y
0370: 20 ED FD  83             JSR COUT
0373: C8        84             INY
0374: D0 F5     85             BNE PRINT
0376: 60        86   ENDE      RTS
                87   ;
0377: 8D 84     88   TEXT      HEX 8D84       ;diverse Texte
0379: CF D0 C5  89             ASC "OPEN"     ;für PRINT
037D: 00 8D 84  90             HEX 008D84
0380: D7 D2 C9  91             ASC "WRITE"
0385: 00 8D 84  92             HEX 008D84
0388: C3 CC CF  93             ASC "CLOSE"
038D: 8D 00     94             HEX 8D00
038F: AE CC     95             ASC ".L"
0391: 8D 00     96             HEX 8D00
0393: 8D 84     97             HEX 8D84
0395: CD CF CE  98             ASC "MONO"
0399: 8D 00     99             HEX 8D00
                100  ;
039B: 09 80     101  AF        ORA #$80       ;Bit 7 setzen
039D: C9 8C     102            CMP #$8C       ;FF?
039F: F0 03     103            BEQ CLOSE      ;Ja, zu Ende
03A1: 4C F0 FD  104            JMP COUT1      ;ausgeben
                105  ;
03A4: A9 F0     106  CLOSE     LDA #<COUT1    ;ASSESSOR-Filer
03A6: 85 36     107            STA CSWL       ;ausklinken
03A8: A9 FD     108            LDA #>COUT1
03AA: 85 37     109            STA CSWH
03AC: 20 EA 03  110            JSR CONNECT    ;DOS informieren
03AF: A0 0F     111            LDY #$0F       ;File schließen
03B1: 20 6B 03  112            JSR PRINT
03B4: A2 F8     113            LDX #$F8
03B6: 9A        114            TXS            ;Stack init.
03B7: 4C D0 03  115            JMP DOSWRM     ;Warmstart, Ende
```

** ENDE **

6. IDUS

Der Debugger und Simulator

Einführung

IDUS ist ein interaktiver Debugger und Simulator für den Apple II. Er soll Ihnen einerseits helfen, Maschinenprogramme zu verstehen, andererseits ist IDUS ein unersätzliches Hilfsmittel bei der Fehlersuche. Mit seinen vielfältigen und mächtigen Möglichkeiten läßt Sie IDUS hinter die Kulissen Ihres Rechners schauen. Eine informative Grafik gibt Ihnen Einblicke, die Ihnen das Verständnis des Apple und der auf ihm laufenden Assemblerprogramme wesentlich erleichtern. Sie können „sehen", was abläuft, und lüften so den Schleier des Geheimnisvollen. Mit IDUS lernen Sie die Assemblerprogrammierung wesentlich leichter, da Ihre Vorstellungskraft angeregt und nicht nur einfach Ihr Gedächtnis strapaziert wird.
Wohl jeder mußte die leidvolle Erfahrung machen, daß Maschinenprogramme selten auf Anhieb so laufen wie geplant. Das Übersehen nur einer Kleinigkeit, oft nur ein nicht zurückgesetztes Carrybit, bringt den wohlgeplanten Ablauf durcheinander, ein „Bug" ist im Programm. IDUS ermöglicht Ihnen, Fehler schnell zu lokalisieren und zu berichtigen, und erhöht so Ihre Produktivität und Freude, denn nichts ist frustrierender als die stundenlange Suche nach einem falschen Byte.
IDUS gehört zu den leistungsfähigsten Programmen seiner Art. Trotzdem ist seine Bedienung durch die weitgehende Verwendung der Menütechnik denkbar einfach. Mit etwas Übung werden Sie bald Ihren Apple kontrollieren.

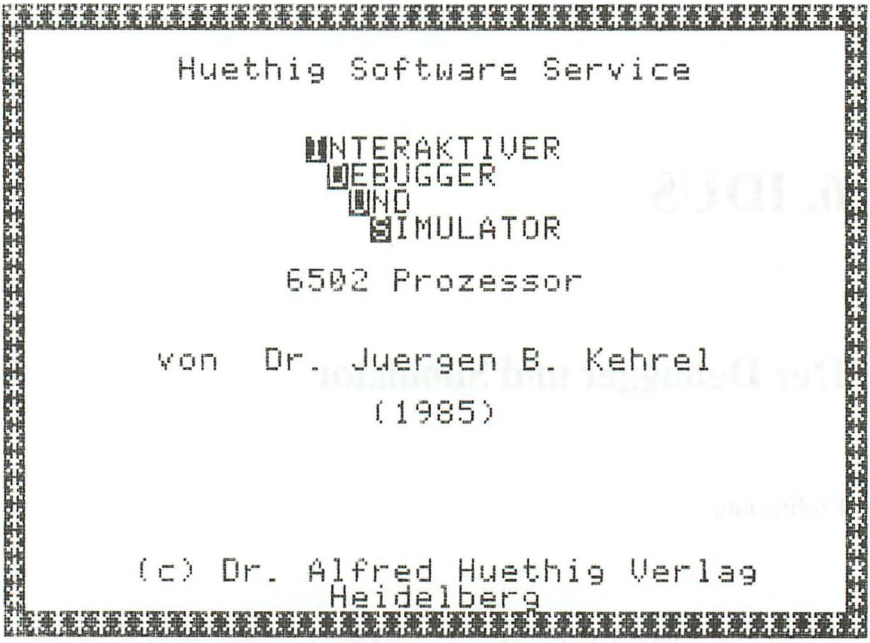


Abb. 38: Titel von Idus

Systemvoraussetzungen

IDUS benötigt 48K RAM sowie Applesoft im ROM und einen Autostart-Monitor. Damit ist IDUS auf praktisch allen Apple IIplus, IIe und IIc lauffähig. Eine Language-Card oder erweiterte 80-Zeichenkarte wird nicht genutzt, auch wenn sie vorhanden ist. Um IDUS zu laden, benötigen Sie ein Diskettenlaufwerk mit DOS 3.3 als Betriebssytem. IDUS modifiziet das DOS im Speicher, um z.B. Fehlermeldungen auf die Grafikseite umzuleiten. Deshalb ist IDUS mit den meisten DOS-Varianten wie Diversi-DOS usw. **nicht** lauffähig. Auch unter ProDOS können Sie IDUS grundsätzlich nicht voll verwenden. Die Ausgabe erfolgt wahlweise auf der ersten oder zweiten hochauflösenden Grafikseite. 80-Zeichhenkarten (VIDEX, Apple usw.) können direkt aus IDUS nicht eingeschaltet werden.
Alle Kommandos werden in Großbuchstaben erwartet. Schalten Sie deshalb bei einem IIe oder IIc die Feststelltaste ein.

IDUS ist für den vollen Befehlssatz des 6502- und des 65C02-Prozessors vorbereitet. Die neuen Befehle des 65C02-Prozessors werden von IDUS aber nur dann verarbeitet, wenn Ihr Rechner mit diesem Prozessor ausgestattet ist. Auf einem Apple IIplus oder IIe sind keine veränderten ROMs erforderlich. (Die bisherige Version des neuen IIe mit 65C02 enthält kurioserweise keinen 65C02-Disassembler im ROM. IDUS disassembliert aber auch hier korrekt, da die notwendigen Tabellenerweiterungen in IDUS enthalten sind.)

Der Start von IDUS

IDUS kann direkt von der Diskette mit „BRUN IDUS" gestartet werden und meldet sich mit einer Titelgrafik, die einen Moment oder bis Sie eine Taste drücken stehen bleibt. Danach sehen Sie die Simulatorgrafik. Grundsätzlich wird zunächst HGR2 zur Anzeige benutzt. Sie können nachträglich zwischen beiden Grafikseiten hin- und herschalten.
Die Kaltstartadresse (vollständige Initialisierung) von IDUS ist $6000. Mit „6003G" aus dem Monitor können Sie einen Warmstart durchführen, der keine Variablen von IDUS löscht.
IDUS zeigt Ihnen in einem Dialogfenster alle möglichen Befehle an (Rollmenü).

Die Simulatorgrafik

Die normale Anzeige von IDUS ist die Simulatorgrafik (Abb.). Sie zeigt von links nach rechts:
a) das X-Register binär und hexadezimal
b) das Y-Register binär und hexadezimal
c) den „Adressenzähler" des 6502
d) einen Speicherausschnitt, der immer auf die gerade bearbeiteten Bytes zeigt (mit ADR gekoppelt)
e) den Akkumulator binär, hexadezimal und als ASCII-Zeichen
f) den Program-Counter des Prozessors
g) den disassemblierten aktuellen Befehl (mit ADR gekoppelt)
h) das Dialogfenster (s.u.)
i) den Prozessor-Status binär
j) den Stackpointer des Prozessors
k) einen Stackausschnitt (mit dem Stackpointer gekoppelt)
l) 6 beliebig wählbare Speicherstellen in RAM oder ROM

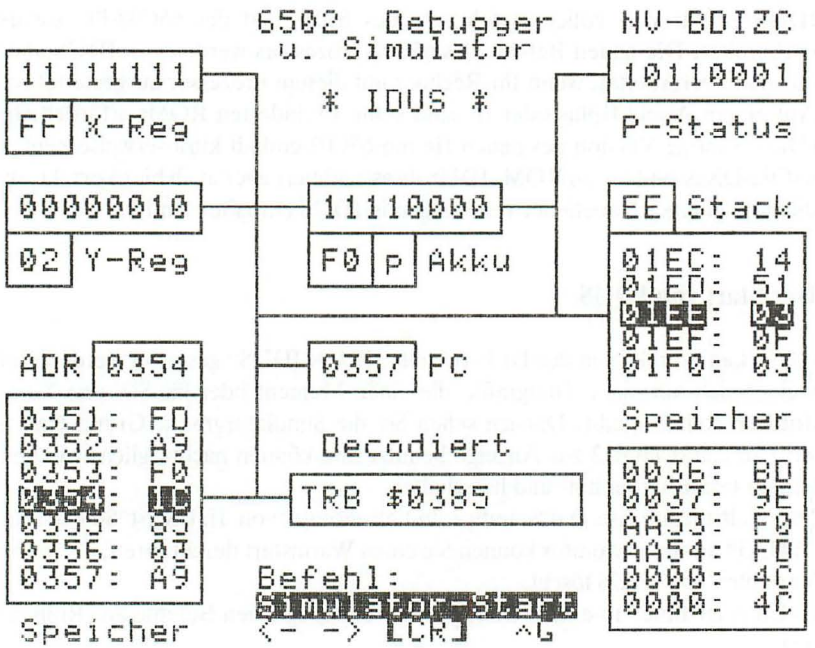


Abb. 39: Simulatorgrafik

Das Dialogfenster

Da wenig freier Raum in der Simulatorgrafik zur Verfügung steht, besitzt IDUS ein kompaktes Rollmenü im Dialogfenster. Mit der Rechts- bzw. Linkspfeiltaste können Sie sich die einzelnen Optionen anzeigen lassen. Wollen Sie einen (angezeigten) Punkt auswählen, so drücken Sie die <RETURN>-Taste. Das Rollmenü ist „endlos", so daß Sie mit beiden Pfeilen alle Möglichkeiten erreichen können. Haben Sie eine Option versehentlich gewählt oder sich nachträglich anders entschieden, bringt Sie <ESC> zurück. Als Sonderfunktion können Sie immer auch <CTRL>-G eingeben (Control-Taste und „G" gleichzeitig drücken) und damit einen Warmstart auslösen. Dies ist für den Fall vorgesehen, daß das „simulierte" Programm die Grafik abschaltet oder verunstaltet. Nach dem Ausstieg aus dem Simulator (siehe dort) können Sie <CTRL>-G eingeben. Wenn IDUS noch intakt ist, bekommen Sie immer wieder die Simulatoranzeige.

Simulator-Start

Dies ist der Einstieg in den Simulator. Sie werden nach der (hexadezimalen) Startadresse gefragt. Sie können beliebig viele Stellen eingeben, ausgewertet werden immer nur die letzten vier vor dem abschließenden <RETURN> (max. FFFF).

Eingabe	Ergebnis
3	0003
33	0033
123	0123
1234	1234
123456	3456

Wenn Sie nur <RETURN> ohne Adresse eingeben, startet IDUS an der Adresse, die momentan im Program-Counter („PC") angezeigt wird. Dies ist eine bequeme Möglichkeit, nach Unterbrechungen an der alten Stelle fortzufahren.

Akku aendern

Sie könnnen jeden beliebigen (hexadezimalen) Wert in den Akkumulator geben. Ausgewertet werden nur 2 Stellen (max. FF).

X-Reg aendern

Auch das X-Register kann von Ihnen Werte zwischen 00 und FF zugewiesen bekommen.

Y-Reg aendern

Das Y-Register steht nicht schlechter da als das X-Register:
00 bis FF sind erlaubt.

Prozessorstatus

Beim Prozessorstatus ist jedes Bit wichtig. Auch hier können Sie jeden Wert laden, um bestimmte Situationen zu erzeugen, z.B. „was wäre, wenn jetzt das Carrybit gesetzt wäre" usw.

Stackregister

Beim Kaltstart wird der Stackpointer (= Stackregisterzeiger) auf FD gesetzt und nach 01FF und 01FE eine „sichere" Stoppadresse geschrieben (s. Spezielles). Seien Sie vorsichtig mit willkürlichen Veränderungen des Stackpointers, da dadurch häufig wichtige Daten für den Prozessor verlorengehen. Der Bereich von 00 bis FF ist zulässig.

Unterbrechungen

Eine bequeme Möglichkeit, den Ablauf der Simulation zu kontrollieren, ist das Setzen von künstlichen Unterbrechungen (Software Breakpoint). Immer, wenn der Program-Counter einen der beiden bestimmbaren Unterbrechungswerte erreicht, stoppt IDUS und kehrt zum Rollmenü zurück. Der Program-Counter muß den Wert, der zwischen 0000 und FFFF liegen darf, genau erreichen. Wird eine Routine im Schnellgang (siehe dort) ausgeführt, wirken die Unterbrechungen **nicht**. Die angezeigte Unterbrechungsadresse bleibt erhalten, wenn Sie nur <RETURN> eingeben. <ESC> führt Sie direkt ins Rollmenü.
Unterbrechungen sind besonders nützlich, wenn Sie nur an dem Zustand einer Stelle interessiert sind und nicht die ganze Zeit zusehen wollen, bis die Simulation dort angekommen ist.

Speicheranzeige

Sie können sechs beliebige (zwischen 0000 und FFFF) Speicherstellen ständig in einem Fenster anzeigen lassen, um z.B. Zero-Page Zeiger „im Auge" zu haben oder Zählbytes zu verfolgen.
Seien Sie extrem vorsichtig mit Werten von C000 bis C0FF, da hier viele „Softswitches" des Apple liegen, die durch die dauernd aktualisierte Anzeige ständig geschaltet werden und dann den Rechner in einen unkontrollierbaren Zustand versetzen.
Wenn Sie Speicheradressen auswählen, werden diese immer der Reihe nach abgefragt. Wollen Sie Adressen unverändert lassen (überspringen), geben Sie nur <RETURN> ein. <ESC> bricht die gesamte Eingabe ab.

Schnellgang

Oft ist es mühselig, auch ROM-Routinen mit zu simulieren (z.B. COUT), wenn Sie eigentlich nur an Ihrem Code interessiert sind.

Hier hilft Ihnen der Schnellgang. Jedes JSR oder JMP zu einer Adresse oberhalb der von Ihnen gewählten Grenze (z.B. F800 für den Monitor) läßt die angesprungene Routine bis zum abschließenden RTS mit der vollen Geschwindigkeit des 6502/65C02 ablaufen. Setzen Sie die Grenze auf FFFF, ist der Schnellgang abgeschaltet.
Wenn Sie die Grenze unter DOS legen ($9D00 bei DOS 3.3), können Sie auch die extrem zeitkritischen Diskettenoperationen mit eingeschaltetem Simulator ausführen. Nach dem Diskettenzugriff erhalten Sie das „Kommando" zurück und können den weiteren Programmverlauf verfolgen.

Apple-Monitor

So gelangen Sie einfach in den Monitor, um beispielsweise größere Speicherbereiche „am Stück" zu disassemblieren, Code zu verschieben, Speicherstellen zu verändern usw. Auch Diskettenbefehle wie „CATALOG" können Sie hinter dem Sternchen eingeben. Mit <CTRL>-Y kommen Sie zurück nach IDUS.

File laden

Der Binärfile (Typ B), dessen Namen Sie angeben, wird von der Diskette geladen. IDUS zeigt Ihnen die Adresse an, ab der der File im Speicher beginnt. Nach einem beliebigen Tastendruck verschwindet diese Anzeige. Tritt ein DOS-Fehler auf, wird die Fehlermeldung in das Dialogfenster eingeblendet, bis Sie eine Taste drücken.
IDUS überprüft nicht im vorhinein, wohin der File geladen wird. Sie sind selbst dafür verantwortlich, daß nicht Teile von IDUS überschrieben werden.

Ausgabe Seite

Sie wählen, ob die Simulator-Anzeige auf HGR1 oder HGR2 erfolgt. Dadurch können Sie auch Programme simulieren, die selbst eine Seite der hochauflösenden Grafik benutzen.

Programmende

Hierzu ist wenig zu sagen: Sie gelangen zurück nach Applesoft.

Der Simulator

Der Simulator führt ab der von Ihnen angegebenen Adresse den Code aus. Ein Teil wird echt im Prozessor verarbeitet, ein anderer Teil nur von Routinen des IDUS.
Auch hier stehen Ihnen wieder einige Kommandos zur Verfügung.

<ESC>

Escape bricht die Simulation ab und führt Sie ins Rollmenü zurück.

Pfeiltasten

Betätigen Sie eine der Pfeiltasten, wird im Dialogfenster die augenblickliche Geschwindigkeit angezeigt, die Simulation stoppt. Mit der Rechts- bzw. Linkspfeiltaste können Sie die Geschwindigkeit verändern. 00 ist die schnellste, FF die langsamste Ausführung (ca. 10 Befehle/Sekunde bis 1 Befehl in 12 Sekunden). Der angezeigte Wert wird durch <RETURN> ausgewählt.
Von 01 bis FF wird jeder Maschinenbefehl in zwei Teilschritten angezeigt, nur bei 00 ist jeder Befehl ein Schritt.

Leertaste

Wenn Sie die Leertaste drücken, hält die Simulation an. Jeder weitere Druck auf die Leertaste führt einen (Teil-)Schritt weiter. Nach <RETURN> wird wieder der Dauerlauf begonnen.

Anzeigeumschaltungen

Wenn Sie mitverfolgen wollen, wie Ihr Programm auf eine der Text- oder Grafikseiten schreibt, können Sie die Simulatoranzeige aus- und die gewünschte Seite einschalten. Die Umschaltung funktionert nur bei Dauerlauf.

<CTRL>-T schaltet die Textseiten ein
<CTRL>-G schaltet die hochauflösende Grafik ein
<CTRL>-L schaltet die niedrigauflösende Grafik ein

Nach dem Drücken der Control-Taste hält der Simulator, bis Sie durch „1" oder „2" die jeweils 1. oder 2. Anzeigeseite aufrufen. Die Eingabe einer Zahl ist **immer** notwendig, um den Simulator wieder zu starten.

Ton

Damit Sie die Simulation akustisch registrieren können, auch wenn die Anzeige ausgeschaltet ist, wird bei jedem (65C02) oder jedem zweiten (6502) ausgeführten Befehl ein „Tick" im Lautsprecher erzeugt. So bekommen Sie mit, wenn etwas „hängt", auch wenn Sie es gerade nicht sehen. Wenn Ihnen das „Tick,tick,tick" auf die Nerven geht, können Sie es mit <CTRL>-Q sowohl an- als auch ausschalten.

Spezielles

Betätigen Sie die <RESET>-Taste nur, wenn es anders nicht mehr geht.
Da IDUS das DOS manipuliert, sollten Sie vor dem Start anderer Programme und speziell vor dem Initialisieren von Disketten neu „booten".
IDUS benutzt die normale Zero-Page, verhält sich jedoch insoweit „transparent", als die vorgefundenen Werte jeweils gerettet und nachher wieder zurückgeschrieben werden. Deshalb sollte es keine Zero-Page Konflikte zwischen IDUS und Ihrem simulierten Programm geben.
Das simulierte Programm benutzt einen eigenen Stack im normalen RAM, so daß Sie vom Monitor aus den Stackinhalt nicht verändern können.
Maschinenprogramme enden häufig mit einem „RTS", das sie zu dem System zurückführt, das beim Start aktiv war (Monitor oder Applesoft). In der Simulation zieht dieses „RTS" eine zufällige Adresse vom Stack, so daß eine Vorhersage, wohin der Sprung führt, nicht möglich ist (häufig Systemabsturz). Um dem vorzubeugen, legt IDUS beim Kaltstart eine Rücksprungadresse auf den Stack, die auf ein $00 (BRK) führt und damit das Programm abfängt. Wenn Sie vor jeder Simulation den Stackpointer auf FD zurücksetzen und die Bytes $01FF und $01FE unverändert sind, bleibt diese Sicherung erhalten.
Auf einem Apple IIe ist es in den meisten Fällen nicht möglich, Maschinencode auf Steckkarten in den Slots zu simulieren, da dieser adressenmäßig parallel zu Teilen des ROMs liegt und der IIe in der Simulation nicht korrekt diese ROMs aus- und die Karten-ROMs einblendet.

Speicherbelegung

IDUS benötigt den RAM-Bereich von $6000 bis $7860 sowie wahlweise die HGR-Seite 1 ($2000-$3FFF) oder 2 ($4000-$5FFF).
Ferner muß immer der Monitor erreichbar sein. Beim Kalt- oder Warmstart sowie bei Tastatureingaben, die mehr als 1 Zeichen lang sind (außer Anzeigeumschaltungen), muß außerdem Applesoft lesbar sein. Es ist deshalb nicht möglich, die Language Card einzuschalten, wenn diese nicht mindestens eine Monitor-Kopie enthält.

Anhang 1: Hex-Dez Umwandlung

	Vierte Ziffer	Dritte Ziffer	Zweite Ziffer	Erste Ziffer
0	0	0	0	0
1	4096	256	16	1
2	8192	512	32	2
3	12288	768	48	3
4	16384	1024	64	4
5	20480	1280	80	5
6	24576	1536	96	6
7	28672	1792	112	7
8	32768	2048	128	8
9	36864	2304	144	9
A	40960	2560	160	10
B	45056	2816	176	11
C	49152	3072	192	12
D	53248	3328	208	13
E	57344	3584	224	14
F	61440	3840	240	15

Beispiel: $711 = 1792
16
\+ 1
1809

Anhang 2: ASCII-Tabelle

NUL	@ §	00	00000000	000	@ §	40	01000000	064	@ §	80	10000000	128	@ §	C0	11000000	192
	A	01	00000001	001	A	41	01000001	065	A	81	10000001	129	A	C1	11000001	193
	B	02	00000010	002	B	42	01000010	066	B	82	10000010	130	B	C2	11000010	194
	C	03	00000011	003	C	43	01000011	067	C	83	10000011	131	C	C3	11000011	195
EOT	D	04	00000100	004	D	44	01000100	068	D	84	10000100	132	D	C4	11000100	196
	E	05	00000101	005	E	45	01000101	069	E	85	10000101	133	E	C5	11000101	197
	F	06	00000110	006	F	46	01000110	070	F	86	10000110	134	F	C6	11000110	198
BELL	G	07	00000111	007	G	47	01000111	071	G	87	10000111	135	G	C7	11000111	199
←	H	08	00001000	008	H	48	01001000	072	H	88	10001000	136	H	C8	11001000	200
TAB	I	09	00001001	009	I	49	01001001	073	I	89	10001001	137	I	C9	11001001	201
↓	J	0A	00001010	010	J	4A	01001010	074	J	8A	10001010	138	J	CA	11001010	202
↑	K	0B	00001011	011	K	4B	01001011	075	K	8B	10001011	139	K	CB	11001011	203
FF	L	0C	00001100	012	L	4C	01001100	076	L	8C	10001100	140	L	CC	11001100	204
RTN	M	0D	00001101	013	M	4D	01001101	077	M	8D	10001101	141	M	CD	11001101	205
SO	N	0E	00001110	014	N	4E	01001110	078	N	8E	10001110	142	N	CE	11001110	206
SI	O	0F	00001111	015	O	4F	01001111	079	O	8F	10001111	143	O	CF	11001111	207
	P	10	00010000	016	P	50	01010000	080	P	90	10010000	144	P	D0	11010000	208
XON	Q	11	00010001	017	Q	51	01010001	081	Q	91	10010001	145	Q	D1	11010001	209
	R	12	00010010	018	R	52	01010010	082	R	92	10010010	146	R	D2	11010010	210
XOFF	S	13	00010011	019	S	53	01010011	083	S	93	10010011	147	S	D3	11010011	211
	T	14	00010100	020	T	54	01010100	084	T	94	10010100	148	T	D4	11010100	212
→	U	15	00010101	021	U	55	01010101	085	U	95	10010101	149	U	D5	11010101	213
	V	16	00010110	022	V	56	01010110	086	V	96	10010110	150	V	D6	11010110	214
	W	17	00010111	023	W	57	01010111	087	W	97	10010111	151	W	D7	11010111	215
	X	18	00011000	024	X	58	01011000	088	X	98	10011000	152	X	D8	11011000	216
	Y	19	00011001	025	Y	59	01011001	089	Y	99	10011001	153	Y	D9	11011001	217
	Z	1A	00011010	026	Z	5A	01011010	090	Z	9A	10011010	154	Z	DA	11011010	218
ESC	[Ä	1B	00011011	027	[Ä	5B	01011011	091	[Ä	9B	10011011	155	[Ä	DB	11011011	219
	\ Ö	1C	00011100	028	\ Ö	5C	01011100	092	\ Ö	9C	10011100	156	\ Ö	DC	11011100	220
	] Ü	1D	00011101	029	] Ü	5D	01011101	093	] Ü	9D	10011101	157	] Ü	DD	11011101	221
	↑	1E	00011110	030	↑	5E	01011110	094	↑	9E	10011110	158	↑	DE	11011110	222
	_	1F	00011111	031	_	5F	01011111	095	_	9F	10011111	159	_	DF	11011111	223
		20	00100000	032	'	60	01100000	096		A0	10100000	160	'	E0	11100000	224
	!	21	00100001	033	a	61	01100001	097	!	A1	10100001	161	a	E1	11100001	225
	"	22	00100010	034	b	62	01100010	098	"	A2	10100010	162	b	E2	11100010	226
	#	23	00100011	035	c	63	01100011	099	#	A3	10100011	163	c	E3	11100011	227
	$	24	00100100	036	d	64	01100100	100	$	A4	10100100	164	d	E4	11100100	228
	%	25	00100101	037	e	65	01100101	101	%	A5	10100101	165	e	E5	11100101	229
	&	26	00100110	038	f	66	01100110	102	&	A6	10100110	166	f	E6	11100110	230
	'	27	00100111	039	g	67	01100111	103	'	A7	10100111	167	g	E7	11100111	231
	(	28	00101000	040	h	68	01101000	104	(	A8	10101000	168	h	E8	11101000	232
	)	29	00101001	041	i	69	01101001	105	)	A9	10101001	169	i	E9	11101001	233
	*	2A	00101010	042	j	6A	01101010	106	*	AA	10101010	170	j	EA	11101010	234
	+	2B	00101011	043	k	6B	01101011	107	+	AB	10101011	171	k	EB	11101011	235
	,	2C	00101100	044	l	6C	01101100	108	,	AC	10101100	172	l	EC	11101100	236
	–	2D	00101101	045	m	6D	01101101	109	–	AD	10101101	173	m	ED	11101101	237
	.	2E	00101110	046	n	6E	01101110	110	.	AE	10101110	174	n	EE	11101110	238
	/	2F	00101111	047	o	6F	01101111	111	/	AF	10101111	175	o	EF	11101111	239
	0	30	00110000	048	p	70	01110000	112	0	B0	10110000	176	p	F0	11110000	240
	1	31	00110001	049	q	71	01110001	113	1	B1	10110001	177	q	F1	11110001	241
	2	32	00110010	050	r	72	01110010	114	2	B2	10110010	178	r	F2	11110010	242
	3	33	00110011	051	s	73	01110011	115	3	B3	10110011	179	s	F3	11110011	243
	4	34	00110100	052	t	74	01110100	116	4	B4	10110100	180	t	F4	11110100	244
	5	35	00110101	053	u	75	01110101	117	5	B5	10110101	181	u	F5	11110101	245
	6	36	00110110	054	v	76	01110110	118	6	B6	10110110	182	v	F6	11110110	246
	7	37	00110111	055	w	77	01110111	119	7	B7	10110111	183	w	F7	11110111	247
	8	38	00111000	056	x	78	01111000	120	8	B8	10111000	184	x	F8	11111000	248
	9	39	00111001	057	y	79	01111001	121	9	B9	10111001	185	y	F9	11111001	249
	:	3A	00111010	058	z	7A	01111010	122	:	BA	10111010	186	z	FA	11111010	250
	;	3B	00111011	059	{ ä	7B	01111011	123	;	BB	10111011	187	{ ä	FB	11111011	251
	<	3C	00111100	060	| ö	7C	01111100	124	<	BC	10111100	188	| ö	FC	11111100	252
	=	3D	00111101	061	} ü	7D	01111101	125	=	BD	10111101	189	} ü	FD	11111101	253
	>	3E	00111110	062	~ ß	7E	01111110	126	>	BE	10111110	190	~ ß	FE	11111110	254
	?	3F	00111111	063	DEL	7F	01111111	127	?	BF	10111111	191	DEL	FF	11111111	255

Anhang 3: Bildschirmdarstellung 1. Zeichensatz

Zeichen	Darstellung		
	Nor	Fls	Inv
	A0	60	20
!	A1	61	21
“	A2	62	22
#	A3	63	23
$	A4	64	24
%	A5	65	25
&	A6	66	26
’	A7	67	27
(	A8	68	28
)	A9	69	29
*	AA	6A	2A
+	AB	6B	2B
,	AC	6C	2C
–	AD	6D	2D
.	AE	6E	2E
/	AF	6F	2F
0	B0	70	30
1	B1	71	31
2	B2	72	32
3	B3	73	33
4	B4	74	34
5	B5	75	35
6	B6	76	36
7	B7	77	37
8	B8	78	38
9	B9	79	39
:	BA	7A	3A
;	BB	7B	3B
<	BC	7C	3C
=	BD	7D	3D
>	BE	7E	3E
?	BF	7F	3F

Zeichen	Darstellung		
	Nor	Fls	Inv
@ §	C0	40	00
A	C1	41	01
B	C2	42	02
C	C3	43	03
D	C4	44	04
E	C5	45	05
F	C6	46	06
G	C7	47	07
H	C8	48	08
I	C9	49	09
J	CA	4A	0A
K	CB	4B	0B
L	CC	4C	0C
M	CD	4D	0D
N	CE	4E	0E
O	CF	4F	0F
P	D0	50	10
Q	D1	51	11
R	D2	52	12
S	D3	53	13
T	D4	54	14
U	D5	55	15
V	D6	56	16
W	D7	57	17
X	D8	58	18
Y	D9	59	19
Z	DA	5A	1A
[Ä	DB	5B	1B
/ Ö	DC	5C	1C
] Ü	DD	5D	1D
^	DE	5E	1E
_	DF	5F	1F

Zeichen	Darstellung Crtl
@	80
A	81
B	82
C	83
D	84
E	85
F	86
G	87
H	88
I	89
J	8A
K	8B
L	8C
M	8D
N	8E
O	8F
P	90
Q	91
R	92
S	93
T	94
U	95
V	96
W	97
X	98
Y	99
Z	9A
[	9B
\	9C
]	9D
^	9E
–	9F

Zeichen IIplus	Zeichen IIe/IIc	Darstellung Nor
		E0
!	a	E1
“	b	E2
#	c	E3
$	d	E4
%	e	E5
&	f	E6
’	g	E7
(	h	E8
)	i	E9
*	j	EA
+	k	EB
,	l	EC
–	m	ED
.	n	EE
/	o	EF
0	p	F0
1	q	F1
2	r	F2
3	s	F3
4	t	F4
5	u	F5
6	v	F6
7	w	F7
8	x	F8
9	y	F9
:	z	FA
;	{ ä	FB
<	| ö	FC
=	} ü	FD
>	~ ß	FE
?	rub	FF

6502 / 65C02-Tabelle

Function	Mnemon.	Akku		Immedi.		Zero-P.		Zero, X		Zero, Y		Absol.		Abs, X		Abs, Y		(Ind, X)		(Ind), Y		Implizit		Relativ		(Ind)		Status
		Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	Op	Cy	N V D I Z C
Load	LDA			A9	2	A5	3	B5	4			AD	4	BD	4*	B9	4*	A1	6	B1	5*					*B2*	*5*	N - - - Z -
	LDX			A2	2	A6	3			B6	4	AE	4			BE	4*											N - - - Z -
	LDY			A0	2	A4	3	B4	4			AC	4	BC	4*													N - - - Z -
Store	STA					85	3	95	4			8D	4	9D	*5*	99	*5*	81	6	91	6					*92*	*5*	- - - - - -
	STX					86	3			96	4	8E	4															- - - - - -
	STY					84	3	94	4			8C	4															- - - - - -
Transfer	TAX																					AA	2					N - - - Z -
	TXA																					8A	2					N - - - Z -
	TAY																					A8	2					N - - - Z -
	TYA																					98	2					N - - - Z -
Stack Ptr	TSX																					BA	2					N - - - Z -
	TXS																					9A	2					- - - - - -
Stack	PHA																					48	3					- - - - - -
	PLA																					68	4					N - - - Z -
Status R.	PHP																					08	3					- - - - - -
	PLP																					28	4					N V D I Z C
Flags	CLC																					18	2					- - - - - 0
	SEC																					38	2					- - - - - 1
	CLI																					58	2					- - - 0 - -
	SEI																					78	2					- - - 1 - -
	CLD																					D8	2					- - 0 - - -
	SED																					F8	2					- - 1 - - -
	CLV																					B8	2					- - - - - -
Jump	JMP											4C	3					*7C*	*6'*							*6C*	*5/6'*	- - - - - -
	JSR											20	6															- - - - - -
Return	RTS																					60	6					- - - - - -
	RTI																					40	6					N V D I Z C
Compare	CMP			C9	2	C5	3	D5	4			CD	4	DD	4*	D9	4*	C1	6	D1	5*					*D2*	*5*	N - - - Z C
	CPX			E0	2	E4	3					EC	4															N - - - Z C
	CPY			C0	2	C4	3					CC	4															N - - - Z C
	BIT			*89*	*2*	24	3	*34*	*4*			2C	4	*3C*	*4*													7 6 - - Z -
Branch	BMI																							30	2+			- - - - - -
N = 0	BPL																							10	2+			- - - - - -
	BEQ																							F0	2+			- - - - - -
Z = 0	BNE																							D0	2+			- - - - - -
	BCS																							B0	2+			- - - - - -
C = 0	BCC																							90	2+			- - - - - -
	BVS																							70	2+			- - - - - -
V = 0	BVC																							50	2+			- - - - - -
Increment	INC	*1A*	*2*			E6	5	F6	6			EE	6	FE	*7/6*													N - - - Z -
	INX																					E8	2					N - - - Z -
	INY																					C8	2					N - - - Z -
Decrement	DEC	*3A*	*2*			C6	5	D6	6			CE	6	DE	*7/6*													N - - - Z -
	DEX																					CA	2					N - - - Z -
	DEY																					88	2					N - - - Z -
Add /Subt.	ADC			69	2	65	3	75	4			6D	4	7D	4*	79	4*	61	6	71	5*					*72*	*5*	N V V - Z C
	SBC			E9	2	E5	3	F5	4			ED	4	FD	4*	F9	4*	E1	6	F1	5*					*F2*	*5*	N V V - Z C
Boolean	AND			29	2	25	3	35	4			2D	4	3D	4*	39	4*	21	6	31	5*					*32*	*5*	N - - - Z -
	ORA			09	2	05	3	15	4			0D	4	1D	4*	19	4*	01	6	11	5*					*12*	*5*	N - - - Z -
	EOR			49	2	45	3	55	4			4D	4	5D	4*	59	4*	41	6	51	5*					*52*	*5*	N - - - Z -
Shift	ASL	0A	2			06	5	16	6			0E	6	1E	*7/6*													N - - - Z C
	LSR	4A	2			46	5	56	6			4E	6	5E	*7/6*													0 - - - Z C
Rotate	ROL	2A	2			26	5	36	6			2E	6	3E	*7/6*													N - - - Z C
	ROR	6A	2			66	5	76	6			6E	6	7E	*7/6*													N - - - Z C
Sonstiges	NOP																					EA	2					- - - - - -
	BRK																					00	7					- - - 1 - -
65C02	*BRA*																							*80*	*3+*			- - - - - -
	PHX																					*DA*	*3*					- - - - - -
	PHY																					*5A*	*3*					- - - - - -
	PLX																					*FA*	*4*					N - - - Z -
	PLY																					*7A*	*4*					N - - - Z -
	STZ					*64*	*3*	*74*	*4*			*9C*	*4*	*9E*	*5*													- - - - - -
	TRB					*14*	*5*					*1C*	*6*															7 6 - - Z -
	TSB					*04*	*5*					*0C*	*6*															7 6 - - Z -
Byteanzahl		1		2		2		2		2		3		3		3		2		2		1		2		2		

* = 1 Takt mehr bei Seitenübergang.
\+ = 2 Takte bei Nicht-Verzweigung, 3 Takte bei Verzweigung, 4 Takte bei Verzweigung mit Seitenübergang.
' = JMP ($HHLL) und JMP ($HHLL, X) je 3 Bytes; passen nicht in Systematik
kursiv = 65C02-Änderungen

Register

&? 197
; 75, 186
& 191
&“ 195
* 185-186
2-Pass-Assembler 183
80-Zeichenkarte, erweiterte 132

A 41
&A 190
Absturz 52
ADC 60, 65, 102, 133
ADD-Funktion 190
Addition 54, 60-61
Addition 16 + 16 Bit 134
Addition 16 + 8 Bit 137, 154
Addition 16 Bit +1 141
Addition 8 Bit + Konst. 145
ADR 110, 204
Adresse 32
Adresse, eigene bestimmen 179
Adressierung, absolute 49, 121
 absolute indirekte 112, 126
 absolute indizierte 122
 absolute indizierte indirekte 127
 Akkumulator- 121, 189
 implizite 54, 121
 indirekte indizierte 123
 indirekte Nullseiten- 127
 indizierte 81
 indizierte indirekte 124
 indizierte Nullseiten- 123
 nachindizierte 125
 Nullseiten- 122
 relative 57, 122
 unmittelbare 49, 121, 188
 vorindizierte 125
Adressierungsart 49, 121, 187
Adressleitungen 132
Adressregister 42
Akkumulator 41
AND 97, 134
Appelsoft-Interpreter 131
APPEND-Funktion 194
ASC 202
ASCII 45, 73
ASCII-Code 74
ASCII-Tabelle 226
ASL 84, 136
&ASM 196
Assembler 36, 182, 184
Assemblercode 37
Assemblerlisting 51
Assembler-Literatur 132
Assemblersprache 36
assemblieren 51
ASSESSOR 50, 182
ASSESSOR-File 192
ASSESSOR-FILER 196, 212
ASSESSOR, Speicherbelegung 209
ASSESSOR, Start 184
ausblenden 97
Ausdruck 187

B 114
Bank 131
Bank-Switching 132
BCC 65, 137
BCD 116
BCS 62, 138
Befehlssatz 6502/65C02 133, 229
Begleitdiskette 11, 13
BELL1 37
BEQ 57, 68, 139
BGE 69, 206
Bildschirm 33
Bildschirmdarstellung 78, 227
Binärmodus 146, 172
Binary-Coded-Decimal 116
Binärzahl 24-25
BIT 100, 120, 139
Bit 23-24, 28
Bit 7 löschen 135, 177
Bit 7 setzen 162, 178
Bitausgabe 95
Bit-Test 135
blinkend 78
BLT 68, 206
BMI 100, 140
BMI 100, 140
BNE 57, 68, 141

Borge-Flagge 63
borgen 63
BPL 74, 100, 142
BRA 96, 142
Branch 57, 122
BREAK 113
Break-Flag (Bit) 113, 119
Break-Point-Debugging 115
BRK 115, 119, 143
BRUN 82
BSP1A 15
BSP1B 15
BSP2A 16
BSP2B 17
BSP3A 17
BSP3B 18
Buchstabe 73
Bug 45, 215
BVC 100, 144
BVS 100, 144
Byte 23-24, 28
Byte, höherwertiges 30, 32
Byte, niederwertiges 30, 32

CALL -151 35, 197
Carry 60, 63, 65, 84, 102, 118
CATALOG 192
CLC 61, 118, 145
CLD 116, 119, 146
CLI 113, 119, 146
CLV 103, 120, 147
CMP 67, 148
Compiler 20
Control-Zeichen 73
COPYA 13
COUT 79
CPX 67, 149
CPY 67, 150
CTRL-E 43
CTRL-G 218
CTRL-Y-Vektor 198

&D 196
DCI 203
DEA 54, 151
Debugger 45, 215
Debugging 115
DEC 54, 151
Decoder 47
Definitionsteil 76
Dekrement 65
Dekrementierung 54
DEL 192
DELETE 194
DEMO1 46
DEMO3 126
Demonstration 15
Demonstration 15
DEX 54, 152
DEY 54, 152
Dezimal-Flagge 116, 119
Dezimalmodus 146, 172
Dezimalsystem 27
Dezimalzahl 25
DFB 204
DFS 90, 204
Dialogfenster 218
digital 21
Disassembler 36
Displacement 57
Distanzbyte 57, 122
Dividend 92
Division 92
Division, Primitiv- 175
Divisionsroutine 16/8 Bit 93
Divisor 92
Dollarzeichen $ 28
Doppelschieben links 168
Doppelschieben rechts 169
Doppelshift 87
DOS 3.3 83, 131
DOS-Varianten 183
Double Shift 168
DUAL1 25
DUAL2 25
Dualsystem 24, 27

&E 193
EBCDIC 73
Editier-Modus 198
Editor 184
einblenden 98
Einer-Komplement 31, 98
END 205
EOR 97-98, 153
EQU 76, 201

EXCLUSIV-ODER 97
EXEC-Funktion 193
Express-Funktion 198

&F 195
Fehlermeldung 37
Fehlermeldungen ASSESSOR 206
Feld 185
Firmware 21
Flagge 41, 56
Flaggen 118
FLS 202
Fragezeichen-Funktion 27, 197
Fragezeichen-Funktion 27, 197

G 35
Ganzzahl (Integer) 30
Garbage 38
gerade/ungerade 85, 161

Hardware 21
Hardware-Break 112
Hauptprogramm 79
HEX 82, 203
Hexadezimalsystem 27
Hex-Dez-Umwandlung 225
HEX-Zahl 27
HGR 33, 131
HGR1 nach HGR2 156
Hi 32, 58
HIMEM: 195
HOME 35, 78, 190

&I 191
I 113
Idee 39
IDUS 45, 215
IDUS, Start 217
immediate 121
INA 54, 154
INC 54, 154
Indexregister 41, 81
Inkrement 65
Inkrementierung 54
INSERT-Funktion 191
Integer 30
Interna ASSESSOR 208
Interrupt 113, 173
Interrupt-disable 113
Interrupt-Disable-Flag 118
Interrupt-Routine 113
INV 202
Invers 78
invertieren 98
INX 54, 155
INY 54, 156
I/O-RAM 132
IRQ 113, 118-119, 146, 173

JMP 83, 157
JSR 79, 107, 158

Kilo-Byte 32
Klammern, spitze 13
Kommando-Modus 190
Kommentar 75, 185
KOMMENTAR-Feld 186
KOMMEX 195, 211
Kommmandointerpreter 184
Kommmandointerpreter 184
Konstante 48
Kopie 13
Korrektur-Funktion 191

&L 191
L 36
LABEL 58, 185
LABEL-Dump 196
LABEL-Feld 185
LABEL-Tabelle 183, 195
Ladevorgang 42
Language Card 131
Lautsprecher 75
LDA 48, 159
LDX 48, 159
LDY 48, 160
LIFO-Speicher 104
LIST-Funktion 191
Listing 80
Lo 32, 58
LOAD 192
LOCK 194
Logik 97
LORES 129
LSR 84, 160
LST 205

M 38
Marke 58, 76
Maschinencode 37, 80
Maschinenprogramm 36
Maske 97, 100
Maskenbyte 98
MASTER CREATE 13
Mauszeichen 78
MEIN PROGRAMM 32
Memory-Dump 36
Menü 109
Microprozessor 21, 41
Mnemonic 36, 185
MNEMONIC-Feld 186
Modulo 92, 135
Monitor 33, 35, 131
Move 38
Multiplikand 89
Multiplikation 84
Multiplikation * 10 85
Multiplikation * 2 87
Multiplikation 16 Bit *2 136
Multiplikation 8 Bit 91
Multiplikator 89
Musik 75

N 69
Negativ-Flagge 69, 120
Negativ-Flagge 69, 120
NEW 190, 192
Nibble 23, 28
Nibble, höherwertiges 28
Nibble, niederwertiges 28
NLS 205
NMI 114, 173
Non-Maskable-Interrupt 114
NOP 111, 161
Normal 78
Notbremse 53
Null 31
Null-Flagge 56, 118
Null-Seite 33, 129
Nummernkreuz # 49, 61
Nur-Lese-Speicher 131

Objektcode 39
ODER 97
Offset 122
OPERAND 185
OPERANDEN-Feld 186
OR 97
ORA 98, 162
ORG 50, 200
Origin 200
Overflow 102, 120

P 41
Patch 161
PC 41
PHA 163
PHP 105, 164
PHX 105, 165
PHY 105, 165
PLA 104, 166
PLP 105, 166
PLX 105, 167
PLY 105, 167
POP 108, 166
POWER-UP-Byte 112
PR# 194
Print-Bits 95
Prioritäten (Interrupt) 115
Problem 39
ProDOS 131
Program Counter 41-42, 107
Programmieren 11
Programmspeicher 129
Programmzähler 41-42, 107
Prompt 35
Prozentzeichen % 26
Prozessorinhalt 43
Prozessorstatus 41
Pseudo-Opcode 50, 200
Pseudo-Opcode 50, 200

Quellcode 39, 80
Quellcode-Format 185, 210

RAM 33-34, 131
Register 41
Register retten 106, 167
Renumber-Funktion 190
RESET 112, 173
Reset-Zyklus 112
Rest 92
Ringtausch 49

ROL 87, 168
Rollmenü 45
ROM 33-34, 131
ROR 87, 169
RTI 114, 119, 170
RTS 51, 79, 108, 170
Rücksprungadresse 107

&S 196
S 41
SAVE 193
SBC 63, 65, 102, 171
Schleife 70, 139-142, 149-150
Schleifenzähler 70, 72
Schreib-Lese-Speicher 131
SEC 63, 118, 172
SED 116, 119, 172
Sedezimalsystem 27
SEI 113, 119, 173
Shift 84
Sign-Bit 31
Simulator 45, 215
Simulator-Bedienung 222
Simulator-Funktionen 219
Simulatorgrafik 217
SOFTEV 112
Software-Break 114
Software-Unterbrechungs-Flagge 119
Sonderzeichen 73
Speicheraufteilung 130
Speicherauszug 36
Speicherbelegung ASSESSOR 209
Speicherbelegung IDUS 223
Speichereinteilung 33
Speichereinteilung 33
Speicher, freier 33, 195
Speicher löschen 152
Speicherplatz 20
Speicherseite 32
Speicherverschiebung 38 ,155
Sprachkarte 131
Sprung, Pseudo- 163
Sprungvektoren 33
Sprungweite 57
STA 48, 173
Stack 33, 104, 129
Stack initialisieren 181
Stackpointer 41-42, 104
Stackpointer versetzen 180
Stacküberlauf 108
Stapel 33, 104, 129
Stapelzeiger 41-42, 104
Stapelzeiger versetzen 180
Status 56
Status Null setzen 166
Statusregister 41-42, 56, 118
String ausgeben 81, 164
Strobe 74
STX 48, 174
STY 48, 174
STZ 116, 175
Subroutine 79
Subtraktion 54, 63-64
Subtraktion 16 - 16 Bit 171
Subtraktion 16 - 8 Bit 151
Subtraktion 16 Bit -1 141
Subtraktion 8 - 8 Bit 172
Suchfunktion 195
System-Monitor 35
Systemvoraussetzungen ASSESSOR 183
Systemvoraussetzungen IDUS 216

&T 194
Takt(e) 21
Tastatur 73
Tastatur 73
Tastaturabfrage 176
Tastatureingabe 74
Tastaturpuffer 33, 129
Tastendruck 100, 140
Tausch X - Y 177
Tausch Y - X 176
TAX 106, 176
TAY 54, 106, 176
Test Bit 6 144-145
Textausgabe 81, 164
TEXTFILE-Funktion 194
Textseite 129
Tonhöhe 75
Transfer 42
TRB 99, 177
TSB 99, 178
TSX 106, 179
TXA 106, 180
TXS 106, 180
TYA 106, 181

&U 198
Überlauf 102
Überlauf-Flagge 120
Übertrag 60, 102
Übertrags-Flagge 18, 60
umklappen 55
UND 97
UNLOCK 194
Unterbrechung 113
Unterbrechungssperre 113, 118
Unterprogramm 79, 107, 158
USER-Funktion 198

V 102
Variable 48
Vektor 129
Vergleich 67, 138
Vergleich 8 Bit 148
Verknüpfung, logische 97
Verschiebung 38
Verschiebung, arithmetische 84
Verschiebung, logische 84
Verschlüsselung 98
Verzweigung, bedingte 57, 120
Verzweigung, erzwungene 142, 144-145, 147
Vorzeichen 30
vorzeichenbehaftet 102
Vorzeichen-Bit 31

Warmstart 83
WARTEN 59
Warteschleife 58
Word 23

X 41
X-Register 41

Y 41
Y-Register 41

Z 56
Zahl 73
Zahlenbasis 27
Zeichenausgabe 152
Zeichendarstellung 45
Zeichenerklärung 133
Zeichenkette 82
Zeiger 210
Zeileneditor 198
Zeilennummer 190
Zero-Flag 56, 118
Zero-Page 33, 129
Zustand, logischer 21
Zweier-Komplement 31, 153

Hüthig

Arne Schäpers

ProDOS-Analyse

Versionen 1.0.1, 1.0.2, 1.1.1

1985, ca. 450 S., kart.,
DM 68,—
ISBN 3-7785-1134-3

„Die ProDOS Analyse" ist die umfangreichste und detaillierteste Darstellung, die jemals ein Apple-Betriebssystem erfahren hat. Wer die „Innereien" von ProDOS bis zum letzten Byte, z. T. bis ins letzte Bit kennenlernen möchte, braucht dieses Buch. Das komplette Betriebssystem (Urlader, MLI, Disk-Driver, RAM-Disk-Driver und Uhr-Routine) mit Ausnahme des BASIC-SYSTEM wird mit umfangreichen Kommentaren und Übersichtstabellen disassembliert. Dabei werden alle bisherigen Versionen von 1.0.1 bis 1.1.1 berücksichtigt. „Die ProDOS Analyse" beschreibt erstmals auch mehrere Programmierfehler, die bis in die neueste Version zu finden sind. Auch die nicht im „Technical Reference Manual" aufgeführten Eigenschaften von ProDOS werden analysiert und beschrieben, z. B. die vertrackten eingebauten Testroutinen zur Identifikation der verschiedenen Apple II Modelle und eventueller Nachbaugeräte. Programmierer, die ProDOS versionsabhängig „patchen" möchten, erhalten hier den genauen Überblick, wo was geändert werden muß, damit dies keine negativen Konsequenzen hat. Durch die minutiöse theoretische Sezierung von ProDOS eröffnen sich völlig neue programmierpraktische Perspektiven.

Dr. Alfred Hüthig Verlag
Im Weiher 10
6900 Heidelberg 1

Apple ProDOS für Aufsteiger, Band 1

Mit ausführlichen Programmbeispielen

von U. Stiehl

1984, 208 S., kart., DM 28,–
ISBN 3-7785-1027-4

Begleitdiskette zu „Apple ProDOS für Aufsteiger", DM 28,–
ISBN 3-7785-1032-0

„Apple ProDOS für Aufsteiger, Band 2",
208 S., kart., DM 28,–,
ISBN 3-7785-1036-3
Begleitdiskette, DM 28,–, ISBN 3-7785-1040-1

ProDOS ist das neue Betriebssystem für den Apple II Plus/IIe. Applesoft-Programmierer, die unter DOS 3.3 gearbeitet haben, werden sich schnell an ProDOS gewöhnen, da ProDOS und DOS 3.3 in dieser Hinsicht weitgehend kompatibel sind. Dagegen müssen Assembler-Programmierer völlig umdenken. Deshalb liegt das Schwergewicht von „Apple ProDOS für Aufsteiger, Band 1" auf der Assemblerprogrammierung und der minutiösen Darstellung der ProDOS-internen Systemadressen, die jedoch auch für Applesoft-Programmierer von großer Bedeutung sind.
Zunächst wird zunächst ein allgemeiner Überblick über das neue „Professional Disk Operation System" gegeben. Im Anschluß daran folgt eine Gegenüberstellung der Geschwindigkeit des Diskettenzugriffs bei DOS und ProDOS. Dann wird die interne Speicherorganisation detailliert beschrieben (Boot-Vorgang, Zero-Page, ProDOS-Vektoren, Basic-System-Puffer, Basic-System-Global-Page, Basic-Command-Handler, I/O-Vektoren, ProDOS-Global-Page, Language-Card-Organisation, Interrupt, Disk-Driver, Reboot-Programm usw.). Ebenso ausführlich wird die externe Speicherorganisation geschildert (Spuren, Sektoren, Blocks, Directory-Struktur, Volume Bit Map, Dateistrukturen usw.). Schließlich wird das MLI (Machine Language Interface) mit zahlreichen praktischen Anwendungsbeispielen erläutert. Insgesamt enthält ProDOS-Buch ca. 70 Seiten mit eigens für dieses Werk entwickelten Programmen. Danach wird das Basic-System technisch für fortgeschrittene Applesoft-Programmierer kurz erläutert.
„ProDOS für Aufsteiger, Band 2" beschreibt ausführlich die Anwendung des ProDOS-BASIC-SYSTEM für Applesoft-Programmierer und enthält darüber hinaus zahlreiche größere Assembler-Anwenderprogramme, die aus Platzgründen in dem ersten Band nicht mehr untergebracht werden konnten.

Hüthig

Ulrich Stiehl

Apple-Assembler

1984, 200 S., 3 Abb., kart.,
DM 34,—
ISBN 3-7785-1047-9

„Apple Assembler" wendet sich an alle, die bereits Anfängerkenntnisse der 6502-Programmierung haben - z. B. aufgrund des Buches „Apple Maschinensprache" - und nunmehr ein Nachschlagewerk für ihren Apple II Plus / IIe / IIc suchen, in dem alle wichtigen ROM-Routinen sowie eine Vielzahl sonstiger Hilfsprogramme in seiner systematischen Form zusammengestellt werden. Insgesamt umfaßt dieses Buch über 40 Utilities, darunter mehrere völlig neuartige Programme wie Double-Lores, Double Hilres, Screen-Format u.a.

Der erste Teil enthält ein Repetorium der wichtigsten Befehle, Adressierungsarten und sonstigen Besonderheiten des 6502.

Im zweiten Teil werden alle Adressen des Monitors zusammengestellt, die für Assembler-Programmierer von Nutzen sein können. Darüber hinaus findet der Leser Unterroutinen für hexadezimale Addition / Subtraktion / Multiplikation / Division, Binär-Hex-ASCII-Umwandlung usw.

Der dritte Teil befaßt sich mit der Speicherverwaltung der Language Card und der IIe-64K-Karte und enthält Move-Programme zum Verschieben von Daten in die und aus der Language Card sowie der 64K-Karte.

Der vierte Teil ist dem Applesoft-ROM gewidmet und listet eine große Anzahl nützlicher Interpreter-Adressen. Bei den Utility-Programmen liegt das Schwergewicht auf Fließkommamathematik einschließlich Print Using.

Der letzte Teil behandelt den Text- und Graphikspeicher. Neben einem professionellen Maskengeneratorprogramm werden auch Routinen zur Double-Lores und Double-Hires-Grafik vorgestellt.

Begleitdiskette zu
„Stiehl, Apple Assembler"
28,— DM
ISBN 3-7785-10487

Dr. Alfred Hüthig Verlag
Im Weiher 10
6900 Heidelberg 1

Arne Schäpers

Hüthig

Bewegte Apple-Graphik

DOS Toolkit-Erweiterungen

1985, 305 S., zahlr. Abb.,
kart., DM 58,—
ISBN 3-7785-1150-5

„Bewegte Grafik, Apple DOS Toolkit Erweiterungen" wendet sich als lehrbuchhafter Kurs an alle, die professionelle hochaufgelöste Grafiken auf dem Apple erzeugen wollen. Der erste Teil beginnt mit einem Abriß des Aufbaus der HGR-Seiten aus der Sicht des Programmierers. Danach wird das Programm HRCG (HI-RES Character Generator, Apple, Inc.) eingehend analysiert und sinnvolle Ergänzungen werden vorgestellt. Schrittweise wird die Nutzung des HRCG erarbeitet bis hin zur beliebigen Bewegung eines statischen Objekts auf einer der HGR-Seiten.

Der zweite Teil baut auf dem ersten auf und führt über die Definition mehrerer Objekte und simultaner Bewegung hin zu einem Arcade-Spiel, das für die meisten käuflichen Action-Spiele in der meisterhaften Grafik als Vorbild dienen kann. Zielgruppe sind Programmierer, die mit Basic keine großen Schwierigkeiten mehr haben und zumindest über praktische Grundkenntnisse in 6502-Assembler verfügen sollten.

Begleitdiskette zu
„Schäpers,
Bewegte Apple-Graphik"
48,— DM
ISBN 3-7785-1290-0

Dr. Alfred Hüthig Verlag
Im Weiher 10
6900 Heidelberg 1

Hüthig

Die Einführung in die Datenbanksprache dBASE II

Wolfgang Eggerichs

dBASE II

Band 1: Einführung

1984, ca. 174 S., kart., DM 39,80
ISBN 3-7785-0986-1

Band 2: Programmierung in dBASE II, ISBN 3-7785-0987-X

In Vorbereitung:

Band 3: Aufbau und Nutzung von Datenbanken mit dBASE II,
ISBN 3-7785-0988-8

Diese Buchreihe befaßt sich mit dem Datenbanksystem dBASE II, einem speziell für Mikrocomputer entwickeltem System. Dieses Datenbanksystem läuft unter den Betriebssystemen CP/M, MP/M, MS-DOS und PC-DOS.

Um den Anfänger den Einstieg in dieses doch recht mächtige Software-Werkzeug zu erleichtern, werden in den beiden ersten aufeinander abgestimmten Bänden jeweils die zu einem bestimmten Leistungsbereich gehörenden Kommandos herausgefiltert und erläutert. Zusätzlich sind kleine Aufgaben integriert, an denen der Leser theoretisch oder/und praktisch seinen Kenntnisstand von dBASE II überprüfen kann. Im dritten Band erfolgt dann eine Darstellung der verschiedensten Einsatzmöglichkeiten von dBASE II.

Der vorliegende 1. Band dieser Einführung befaßt sich mit der reinen Dialogarbeit in dBASE II. In kleinen, methodisch aufeinander abgestimmten Kapiteln erfährt der Leser die wichtigsten dBASE II-Kommandos, um kleine Aufgaben der Datenverwaltung selbständig lösen zu können.

Aus dem Inhalt:

Befehle zum Erfassen, Anzeigen von Listen und Daten · Ändern, Kopieren und Sortieren von Datenbankinhalten · Zeichen(ketten)-Bearbeitung in dBASE II · Arbeiten mit indizierten Datenbanken in dBASE II · Löschen und Ändern von Datensätzen, Dateien und Datenbankstrukturen · Erstellung von Summen und Berichten · Verändern von dBASE II-Systemeinstellungen · Durchführung und Erstellung von Kommando-Dateien · Weitere dBASE II-Anweisungen

Dr. Alfred Hüthig Verlag
Im Weiher 10
6900 Heidelberg 1

Das zweibändige Werk „Apple-Assembler lernen“ ist ein kompletter Kursus über die Assemblerprogrammierung des 6502 und 65C02 auf dem Apple II, der nicht da aufhört, wo andere Einführungen auf „weiterführende Literatur“ verweisen.

Besondere Betonung wird auf die praktische Anwendung gelegt. Deshalb gehört zum Kursus ein vollwertiger 2-Pass Assembler ASSESSOR, der als einer von wenigen die erweiterten Befehle der neuen IIc und IIe Prozessoren 65C02 verarbeitet und der auch lange Programme von mehr als 1000 Zeilen in wenigen Sekunden übersetzt. Zu seinen professionellen Eigenschaften gehören 15 Pseudo-Opcodes, die die Arbeit mit Strings und Tabellen zu einem Kinderspiel werden lassen, und ein Zeileneditor mit viel Komfort. Der interaktive Debugger und Simulator IDUS hilft Ihnen, eigene und fremde Maschinenprogramme zu verstehen. Sie können „sehen“, wie der Prozessor und der Speicher Ihres Computers arbeiten.

Im ersten Band erlernen Sie innerhalb von 35 Lektionen sämtliche Maschinenbefehle des Apple und die wichtigen Grundalgorithmen. Der Umgang mit dem Assembler und dem Simulator wird geübt, und zum Nachschlagen steht eine ausführliche Befehlsbeschreibung mit vielen praktisch nutzbaren Beispielen bereit.

Im zweiten Band wird dann der Umgang mit den eingebauten ROM-Routinen ausführlich besprochen. Grafik, Sound, Stringverwaltung, Fließkommaarithmetik werden ebenso behandelt wie Diskettenoperationen unter DOS 3.3 und ProDOS oder das Zusammenwirken von Applesoft und Assemblerprogrammen.